AD-A059 041 NAVAL SURFACE WEAPONS CENTER WHITE OAK LAB SILVER SP--ETC F/G 9/2
A FIXED-POINT ARITHMETIC COMPILER.(U)
JUL 77 P CRAUN
UNCLASSIFIED NSWC/WOL/TR-77-65 NL

1 OF 2
AD
A059041

NSWC/WOL TR 77-65

LEVEL II

12

A FIXED-POINT ARITHMETIC COMPILER

BY PAUL CRAUN, III

ORDNANCE SYSTEMS DEVELOPMENT DEPARTMENT

1 JULY 1977

NAVAL SURFACE WEAPONS CENTER

Dahlgren, Virginia 22448 • Silver Spring, Maryland 20910

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (14) NSWC/WOL/TR-77-65 ✓	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)* (6) A FIXED POINT ARITHMETIC COMPILER.	5. TYPE OF REPORT & PERIOD COVERED (9) Final rept. ✓
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) (10) Paul Craun, III	8. CONTRACT OR GRANT NUMBER(s) (16) WØ476
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Surface Weapons Center White Oak Silver Spring, Maryland 20910 (17)	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 63254N; WO476; WØ476; U61C;
11. CONTROLLING OFFICE NAME AND ADDRESS (11)	12. REPORT DATE 1 Jul 77
	13. NUMBER OF PAGES 116
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*	15. SECURITY CLASS. *(of this report)* Unclassified (12) 133p.
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release, distribution unlimited

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Fixed-Point Arithmetic Compiler
Macro
Parsing Algorithm
Macro Optimization
Macro Expansion

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

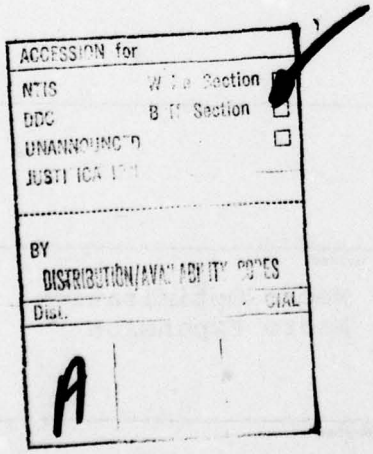
Mini computers used within real time digital systems require mathematical operations to be performed at maximum speed. The most efficient use of their fixed point arithmetic units is made while manipulating fixed point data. Software simulation of floating point mathematics is too slow and requires additional core. A fixed point arithmetic compiler is designed to allow the programmer to express fixed point mathematical processes in → next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

391 596

equation form. Only the description of the original equation arguments is required. All intermediate results of the equation realization are scaled by the compiler to avoid clipping and truncation errors.



NSWC/WOL TR 77-65

A FIXED-POINT ARITHMETIC COMPILER

by

Paul Craun, III

Thesis submitted to the Faculty of the Graduate School
of the University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1972

PREFACE

This report should be of interest to people involved in generating high-level languages that incorporate facilities for manipulating fixed-point data.

Edward C. Whitman

EDWARD C. WHITMAN
By direction

CONTENTS

Chapter		Page
I	INTRODUCTION	5
	A. The Problem	6
	B. The Machine	6
II	THE MACRO APPROACH	9
	A. Compiler Via the Macro Approach	10
	B. Compiler Card Images	11
	C. Syntax of LET, General and Specific Macros	13
III	FIXED-POINT NUMBER SYSTEM	15
	A. Format of Fixed-Point and Floating-Point Numbers	15
	B. Fixed-Point Number Representation	18
	C. Arithmetic Compiler Information Vehicles	22
IV	DETERMINATION OF M AND T	27
	A. Addition and Subtraction M and T Fields	28
	B. Multiplication M and T Fields	30
	C. Division M and T Fields	32
	D. Subroutine Call M and T Fields	33
	E. Equality M and T Fields	37
	F. Exponentiation M and T Fields	38

CONTENTS (Cont.)

Chapter		Page
V	SPECIFIC MACRO REALIZATION	39
	A. Addition and Subtraction Investigation	39
	B. Multiplication Investigation	44
	C. Division Investigation	46
	D. Subroutine Call Investigation	48
	E. Equality Investigation	49
	F. Exponentiation Investigation	50
VI	THE DOPE ASSEMBLER AND THE ARITHMETIC COMPILER	51
	A. Pass 1 and the Parsing Algorithm	51
	B. Parsing Algorithm Theory	52
	C. Pushdown Stacks and Tables	53
	D. Pass 1.5 and Macro Optimization	55
	E. Macro Optimization Theory	55
	F. Partition Concept of Specific Macros	56
	G. Pass 2 and Macro Expansion	58
	H. Macro Expansion Theory	59
	I. Core Requirements for Specific Macros	60
	J. Address Field Coding	62
VII	RESULTS AND CONCLUSIONS	65
	A. Areas of Improvement	65
	B. Useful Features	66
APPENDIX A.	COMPILATION EXAMPLE	67

CONTENTS (Cont.)

Chapter Page

APPENDIX B. PARSING ALGORITHM PRECEDENCE TABLE..... 82

APPENDIX C. SPECIFIC MACRO EXPANSIONS..... 84

SELECTED BIBLIOGRAPHY..... 116

LIST OF FIGURES

Figure

C1 ARITHMETIC COMPILER SETS WITH SUBSETS..... 85

C2 ADDITION SET WITH SUBSETS AND ELEMENTS IN GROUPS..... 86

C3 SUBTRACTION SET WITH SUBSETS AND ELEMENTS IN GROUPS..... 87

C4 MULTIPLICATION, DIVISION, AND EQUALITY SETS WITH SUBSETS AND ELEMENTS IN GROUPS..... 88

C5 SUBROUTINE CALL SET WITH SUBSETS AND ELEMENTS IN GROUPS..... 89

CHAPTER I

INTRODUCTION

This report is an investigation into the design of a fixed point arithmetic compiler for mini computers. The compiler is capable of translating single statement arithmetic equations into assembly language code where the symbolic address variables of the equation refer to single precision two's complement numbers.

The investigation begins with a study of fixed point numbers to define the characteristics that force the mathematical manipulations to be programmer dependent. Once defined, a compiler algorithm is designed to use these characteristics to minimize the need for the programmer to be concerned with binary points, or clipping and truncation errors no matter how complex the equation.

The design approach chosen for equation evaluation is to parse the arithmetic equation into a series of macro statements. Each macro statement contains an indication of which mathematical operation is to be performed, the appropriate symbolic addresses, and the shift parameters derived from the characteristics of each of the operands. The arithmetic compiler contains sets of precoded macros for any possible fixed point mathematical operation. One macro is chosen for each operation in the equation, the symbolic addresses and shift parameters are incorporated, and the symbolic code is entered into the body of the program being assembled. A standardization is achieved in equation evaluation using the macro approach that was never possible with each programmer coding his own realizations.

Other than this overall approach to the design, it is not practical to discuss an arithmetic compiler that will apply to all mini computers when taken as a group. The variety of arithmetic units, existing software, and physical constraints make it necessary to choose a specific machine and implement the macro approach on it as an illustration. The arithmetic unit only directly affects the form of the mathematical macros, whereas the existing software and the physical characteristics of the machine influence the actual coding of the compiler itself. Clearly the less available core and the lack of flexibility of the existing assembler will complicate the generation of the compiler. These two factors are critical considerations as to whether or not it is possible to implement the design on any particular machine.

A. The Problem

Mini computers have seen increased use as modular components within real time digital systems. The machines are programmed to analyze or simulate real world processes by realizing the mathematical models. An arithmetic compiler simplifies the programming operation.

Floating point arithmetic compilers are available in the more sophisticated languages such as FORTRAN and ALGOL, but are not practical in this application because of speed and available core restrictions.

Mini computers have fixed point arithmetic units. Floating point operations have to be software simulated, which greatly increases the execution time of the program, and hinders operation in real time. Software simulation is not only slow, but takes additional core which is not always available. The efficient coding that a fixed point arithmetic compiler could generate would meet the speed and core requirements of real time operation.

A criterion is established to which such a fixed point compiler must conform. It must be incorporated into the existing assembler of the machine such that the output listings and object code of programs assembled before its insertion shall be identical to the listings and object code of the programs assembled afterwards. Where feasible, the standard macros must display maximum speed and minimum length. Optimization must be performed between macros to eliminate unnecessary loading and storing of partial results that would slow down the execution of the entire equation.

B. The Machine

The arithmetic compiler design is implemented on the Digital Simulator and Computer (DISAC). It has the need for a concise method of handling arithmetic equations because of its use in adaptive signal processing and digital simulation. Being a first generation mini computer, its 8-microsecond cycle time is an order of magnitude slower than the state of the art. This further emphasizes the need for efficient coding. The physical and operational characteristics are listed in brief below; reference 1 offers more detail.

1. Two's complement and logical functions are performed in either of the two operational registers, the AC (accumulator) or the MQ (multiplier-quotient). Interregister and from-memory operations can be performed. Both registers have variable length logical shift capability with the full complement of arithmetic shifts being reserved for the MQ alone.

2. Word length is 24 bits with the memory access instructions being able to directly address the entire 4096 words of random access core. Indirect addressing is not necessary.

3. Addition, subtraction, loading and storing, and increment and decrement testing of the three 12-bit hardware index registers are provided for a powerful addressing scheme.

4. DISAC assembly language programs are assembled by the DISAC Object Program Encoder, referred to by its acronym as DOPE. Source programs are entered through the card reader and the images are copied onto magnetic tape unit two. This enables multiple passes through the source deck at tape unit speed. Tape unit two is also the library tape which contains DOPE and all the standard library subroutines.

DOPE produces absolute or relocatable object deck images that are loaded and linked at run time by the resident system of DISAC. Further information on the System and DOPE is available in reference 1. In order to incorporate the arithmetic compiler, the structure of DOPE had to be modified. Distinction between the two versions is made by the prefix 'old' in the discussion to follow.

Old DOPE was a two-pass assembler with each pass being a single long absolute program (3265_8 first pass and 3241_8 second pass). Each occupied essentially the same area of core, the second pass overwriting the first at the appropriate time. The System scanned the library tape on unit two and read the first pass into core. Pass 1 read the source deck from the card reader and performed its function of creating the symbol table and copying the card images also on tape unit two. At the end of the first pass, the System scanned the library tape again and loaded the second pass into core. This action was initiated by the recognition of the END card by pass 1. The second pass read the card images of pass 1 from tape unit two and performed its function by making use of the symbol table. Its output was the image of the object deck and listing on tape unit one.

Any modification attempt of old DOPE would depend heavily on the availability of core and the ease with which changes could be implemented in its program code. The available core in pass 1 is all the memory not dedicated to the System, the first pass, and the symbol table. This is approximately 900 decimal locations. The second pass has slightly more room, 920 locations. The programs that make up the arithmetic compiler must fit in this space or another pass will have to be added. If this is necessary, it would be desirable to have DOPE operate as the three-pass compiler while still remaining a two-pass assembler.

The fact that each pass of DOPE was a single long absolute program made it very difficult to modify. Every change or addition would require reassembly of the entire pass. A new approach was taken; each existing pass was broken down into small functional relocatable subprograms. With this approach, a single subprogram could be modified, assembled, and debugged independently of the rest of the pass. This relocatable version of DOPE requires a different

loading sequence from the System to place it in core during the assembly process. The System no longer loads a single absolute program; instead it scans the library tape for each of the functional subprograms, relocatably loads them into core, then links them into a unified pass. The second pass is loaded in the same manner after the recognition of the END card.

The modifications to old DOPE were made with the result that no change can be observed between the object code and the listing created by old DOPE and the listing and object code generated by the new relocatable version. With the assembler in a form that could be easily modified, the implementation of the arithmetic compiler could begin.

CHAPTER II

THE MACRO APPROACH

With DOPE modified to easily accept additional programs, an investigation was made into different possible ways that a fixed point arithmetic compiler could be implemented. Only two choices were considered. The first is that each mathematical operation in the equation could be implemented with general purpose closed subroutines, one per different operation. The second choice is that each operation could be implemented with a specialized open subroutine implanted in the body of the program.

The closed subroutine approach has two apparent advantages. The first is that only one general purpose 'add' subroutine, for example, would be needed for a program irregardless of how many equations it contained or how many add operations were indicated. The second advantage is that within the body of the program there are only 'calls' to the external subroutine.

There is a subtle point to the 'call' procedure, however, that turns out to be a major disadvantage. It will be shown later that most mathematical operations involve three arguments and three shift parameters. These six pieces of information, or their addresses, would be stored in the body of the program sequentially following the 'call' statement. Therefore, for each mathematical operation, approximately seven words of core would be required for each transfer to the external subroutine.

The second disadvantage is that the closed subroutine has to be general in design. As will be shown later, for example, there are 405 possible ways to add two fixed point numbers and place the result in a third storage location in DISAC. Code must exist in the one closed subroutine to realize all of these possibilities. Just from the sheer bulk of these statistics, it can be concluded that the closed subroutine is going to be lengthy, consisting mainly of bookkeeping algorithms, and be extremely slow.

The open subroutine, or macro approach, uses specialized blocks of code called macros. Each macro consists of code entirely dedicated to performing the mathematical operation. Bookkeeping chores and the deciding of which macro is to be used are determined by the arithmetic compiler at compile time. The disadvantage of having to have an 'add' macro, for example, inserted into the body of the program each time a (+) appears in the equation, is not serious. The average length of all the 405 possible add macros is less than five instructions in length. This is shorter than the calling sequence alone

for the closed subroutine approach.

The macro approach is the only feasible method of attack to choose and still stay within the scope of the problem, to produce optimum code with maximum speed and minimum length.

A. Compiler Via the Macro Approach

The macro approach allows the arithmetic compiler to be functionally divided into three distinct programs: Parsing Algorithm, Macro Optimization, and Macro Expansion. The Parsing Algorithm routine resides in the first pass of DOPE with Macro Expansion in the second. A new pass has to be added between pass 1 and pass 2 to hold Macro Optimization. To indicate its relative position in the compilation process, it is logically named pass 1.5.

A program containing an arithmetic equation is operated on by DOPE under control of the System. The library tape is scanned and all the functional subprograms that make up pass 1, including the Parsing Algorithm, are loaded and linked. This first pass operates as a normal first pass of an assembler on all assembler instructions. It reads the source deck from the card reader and creates an output buffer tape of source card images and first pass bookkeeping statistics. The Parsing Algorithm operates only on the arithmetic equation and creates its own images on the output tape. At the end of pass 1, the library tape is scanned again, and all the functional subprograms of pass 1.5 are loaded and linked.

Macro Optimization of pass 1.5 reads the output buffer tape of pass 1 in search of the images created by the Parsing Algorithm. All other card images are copied onto the output tape of pass 1.5 with only minor bookkeeping changes. The Macro Optimization routine operates primarily on the Parsing Algorithm images, then writes the modified forms on the output buffer tape. At the end of pass 1.5, the library tape is scanned for the third and final time and the pass 2 functional subprograms are loaded and linked.

Pass 2 reads the output buffer tape of pass 1.5 and performs normal second pass functions on all the assembler instruction images. The Macro Expansion routine searches for the modified images of the Macro Optimization routine and expands each into a block of assembler code. At the termination of pass 2, an object deck has been created that consists of all the instructions originally in the source deck plus the instructions inserted by the arithmetic compiler to realize the equation. The accompanying output listing also shows both the original assembler instructions and the compiler created code.

B. Compiler Card Images

The only card image formats of concern to the programmer are those of the input equation and the output listing. In addition to these, there are two other compiler dependent card images, the general macro of the Parsing Algorithm and the specific macro of Macro Optimization. Below is given an explanation of each format.

1. Format of the Input Equation. The input source card has the unique mnemonic, LET, in its instruction field to distinquish it from all other source cards. Any allowable FORTRAN equation that can be derived from the Precedence Table of Appendix B can appear in the variable field. An example of a card with a label is shown below. The variables, or arguments, in the equation are the symbolic addresses of the fixed point data.

```
CLYD   LET  ALFA = BETA + C * (A + GAMA)
```

2. Format of the General Macro. The general macro images that the Parsing Algorithm generates can have one of seven unique mnemonics in the instruction field. The seven are listed below.

General Macro Mnemonic	Operation
ADDM	addition
SUBM	subtraction
MULM	multiplication
DVSM	division
EXPM	exponentiation
EQUM	equality
CALM	subroutine call

The variable field of this image contains a maximum of three arguments which are either original variables of the input equation or symbolic addresses created by the Parsing Algorithm to hold partial results. Examples of typical general macros are shown below. Note the $ delimiter.

```
ADDM A    B     $VOO  $
SUBM ARG1 ARG2  $VO2  $
EQUM $VO3 BETA  $
```

3. Format of the Specific Macro. The specific macro card image generated by the Macro Optimization routine are two-part transformations of the general macros. The first part is the generation of a unique specific macro mnemonic for the instruction field. The scheme converts the second alphabetic character of the general macro mnemonic to a D, E, or F; the third character is changed to a 1, 2, 3, 4, 5, 6, 7, or 8; and the fourth character transforms to a 0, 1, or 2. Therefore an ADDM could be transformed into an AF52, or an MULM could become an MF11. Not all of the above possible specific macro mnemonics are valid; the ones that are realized in the arithmetic compiler are shown in Appendix C.

The second portion of the transformation is performed on the variable field of the image with the addition of shift parameters and the possible substitution of a reserved mnemonic for an argument.

The variable field of the specific macro card image contains a maximum of three arguments and three shift parameters. One or more of the arguments may be replaced by one of the special reserved compiler symbols, AC or MQ, to indicate that the argument is in the AC or MQ respectively. Two examples of possible transformations are shown below.

```
ADDM A    B    $VOO   $ ' AF20 A    B    $MQ$  $YY  1

MULM BETA ALFA $VO3   $ ' ME11 $MQ$ ALFA $AC$  $YY  3 4
```

Note that the $ is expanded to the $YY in-between the argument and the parameter fields. YY is a number from 01 to 99 that indicates the order in which the mathematical operations are performed. They can be used to distinguish and point to individual macros within the equation expansion. If YY was an 03, then the macro is the third one in the macro string. Having a means of referring to an individual macro gives the programmer a way of bypassing or inserting BMT information into the equation on the macro level. This flexibility is explained in the CORR discussion of section IIIC3.

4. Format of the Listing. The assembler instruction mnemonics that appear in the listing as a result of the expansion of the specific macro are indistinguishable from the assembler instructions that were in the source deck. To make it possible to distinguish the code, each macro image and the original LET card are written out on the listing as remarks. This makes it possible to bracket the code that realizes the equation. A brief portion of the program listing is shown below.

Input card in the Source Deck

```
CLYD    LET  A = B + C - D
```

Listing

```
                        LET  A = B + C - D
                        AF30 B    C     $MQ$   $1   1 2 1
0010 20400051 CLYD      LDMQ C
0011 54000001           AQRS 2
0012 20200052           LDAC B
0013 56000002           ALS  1
0014 12500000           AACQ
0015 54000001           AQRS 1
                        SD81 $MQ$ D     $MQ$   $2 0 1 1
0016 20200053           LDAC D
0017 56000001           ALS  1
0020 12540000           SACQ
0021 54000001           AQRS 1
0022 63600050           STMQ A
       .
       .
       .
       .
       .
       .
       .
       .
0050 00342131 A         OCT  00342131
0051 77770302 B         OCT  77770302
0052 00662211 C         OCT  00662211
0053 00000010 D         OCT  00000010
```

C. Syntax of LET, General and Specific Macros

The syntax of the LET statement, general macro, and the specific macro have a designated field structure. The position and widths of each field are identical to those of the normal assembler instructions as explained in reference 1 section 5. Columns 1 through 7 may contain any continuous string of alphanumeric characters of which one of the last four must be a letter; only the last four characters are recognized as the label. Columns 8 through 12 contain the instruction mnemonic. Beginning in column 13 and inclusive through column 72 is the variable field. It is in this field that the three card images have their greatest differences.

The variable field of the LET statement must contain an equation of allowable operators and symbolic addresses. The seven allowable operations are listed in section IIB2. Precedence indicators, the

parentheses, and the index operator, the comma, may also appear. There is no restriction on where within the variable field the equation may begin or how many blank spaces may appear between operator and argument. All arguments on the right side of the equal sign must appear in an ASSN pseudo-op in the program or on the left side of an equal sign of a previous LET. It is optional whether they also appear as label. The single argument on the left of the equal sign does not have to appear in an ASSN statement or as a label. This relationship between the ASSN pseudo-op and the symbolic address argument is explained in section IIIC1.

The variable field of the general macro contains no more than three arguments separated by any number of blank spaces. The arguments may be simple symbolic addresses or any combination of symbolic addresses. Any address field that is allowed in a normal assembler instruction can be an argument. The $ delimiter terminates the argument string.

The variable field of the specific macro is identical to the general macro with the addition of the shift parameter data string and the $YY delimiter. There can be no greater than three shift parameters in the string. Each one is a decimal number separated by any number of blank spaces. Section IIB3 offers examples of both general and specific macro variable fields.

CHAPTER III

FIXED POINT NUMBER SYSTEM

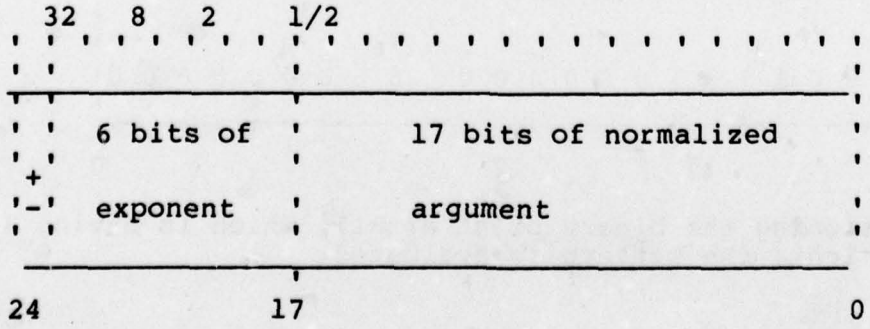
The programmer makes use of certain characteristics of the fixed point numbers when he manually generates code to realize an equation. In order to design a compiler to be as efficient, these characteristics must be made available. A comparison between fixed and floating point numbers is made to emphasize that the compiler complexity is due to the input and use of this additional information. Programmer dependence is therefore reduced to just the initial description of the fixed point numbers, leaving the manipulation of all intermediate results to the compiler.

A. Format of Fixed Point and Floating Point Numbers

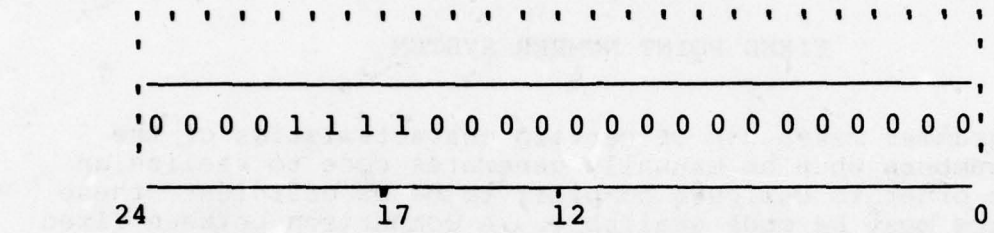
The floating point format is fairly well known because it is very similar to 'scientific notation' in which all numbers are represented as normalized fractions raised to a power of 10. Examples are shown below.

Number	Scientific Notation
235	$.235 * 10^3$
5	$.5 * 10^1$
.0029	$.29 * 10^{-2}$

Assume for the purpose of this discussion that there exists a computer identical to the one described in section IB except that it has a floating point arithmetic unit. The 24-bit floating point numbers of this machine are proposed to have the base 2 format as opposed to the base 10 of scientific notation.



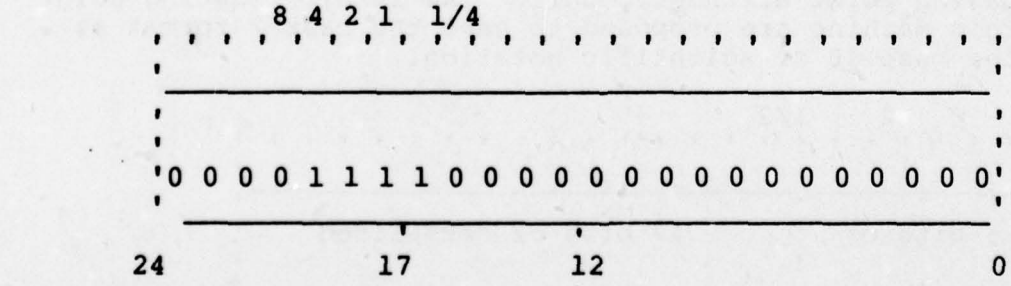
Numbers have an approximate dynamic range of + or -2^{40} (10^{12}) to + or -2^{-23} (10^{-7}) with full 17-bit accuracy on the normalized argument. An important feature of this format is that the pattern of 1's and 0's that make up the number is unique. An example is shown below.



The above pattern represents the number $.5 * 2^7$ which is equal to decimal 64. It is the partitioning of the word into an exponent and a normalized argument that makes this pattern uniquely 64; no programmer interpretation is needed. Mathematical operations with the floating point arithmetic unit produce results in the same unambiguous format.

The fixed point format requires programmer interpretation, or interaction, to make the pattern of 1's and 0's a unique number. Using the standard two's complement weighting notation reviewed in reference 1 section 2.1, and introducing the binary point concept, the programmer can state that the same pattern above is the number 7.5 B17. The dynamic range of a fixed point number is from approximately + or -2^{23} (10^{7}) to + or -2^{-23} (10^{-7}). Bit accuracy is a variable depending on the actual position of the binary point.

The use of the binary point is a method of partitioning the 24-bit word into an integer and a fractional part. All bits to the left of the binary point are integer magnitude bits and all the bits to the right make up the fractional part of the number. An example is shown below.



By positioning the binary point at B17, which is having 17 bits to its right, the pattern is evaluated.

$$\ldots +0*2^{4} +0*2^{3} +1*2^{2} +1*2^{1} +1*2^{0} +1*2^{-1} +0*2^{-2} + \ldots$$

$$\ldots + 0 \quad + 0 \quad + 4 \quad + 2 \quad + 1 \quad + 1/2 \quad + 0 \quad + \ldots$$

$$+ 7.5$$

The ambiguity is immediately obvious. The above pattern could just as easily represent any one of 23 other numbers depending solely on where the programmer positions the binary point. If he positions it at B18, then the above same pattern is evaluated to 3.75 B18.

Programmer interaction is required not only in number interpretation, but also in performing the mathematical operations. Section IV goes into detail on the set up and the interpretation of results for all fixed point mathematical operations.

A simple comparison of addition in both formats is given below to emphasize the degree of difference in the programmer interaction required.

$$235 \qquad .235 * 10^{3}$$

$$5 \qquad .5 * 10^{1}$$

To add the above two floating point numbers, the exponents must be made to coincide. Depending on the scheme built into the floating point adder hardware, the shifting of one of the normalized arguments ($.5*10^{1}$ to $.005*10^{3}$) is done by the adder hardware. The addition is then performed which yields a result of $.240 * 10^{3}$. The programmer does not have to concern himself with the fact that the exponents were not equal before the additon operation, the arithmetic unit took care of it for him.

In the fixed point system, the programmer must align the binary points of his numbers under program control before the hardware addition can take place.

7.5 B17

3.75 B18

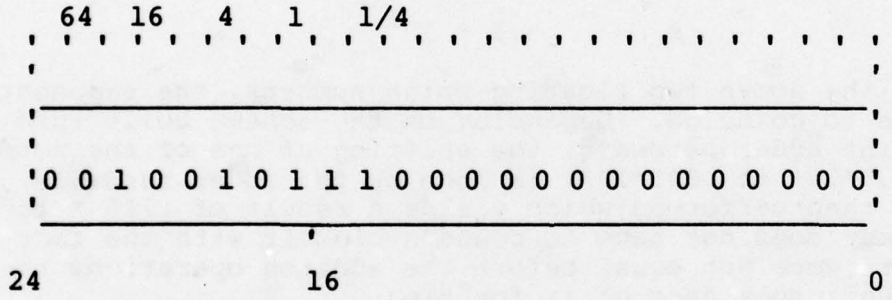
To add the above two fixed point numbers, one of them must be shifted to make them both B18 or both B17. Choosing B18 dictates that 7.5 B17 must be left shifted one position to make it 7.5 B18. The result of the addition yields 11.25 B18 only because the programmer is aware that the binary point is at B18.

For the arithmetic compiler to manipulate the fixed point numbers with the same proficiency as the programmer, it too must have the binary point information. Having exposed the primary restriction that the fixed point number system implies, it remains to implement

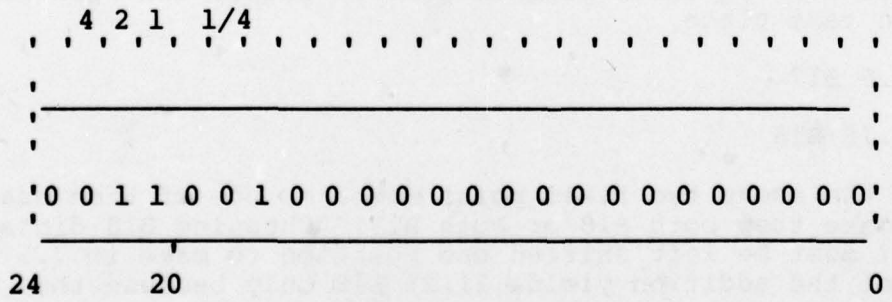
a means of supplying information, such as location of the binary point, to the arithmetic compiler.

B. Fixed Point Number Representation

During any mathematical operation with fixed point numbers, the programmer must be aware of the binary points of the original numbers as well as for all results of mathematical operations. There are two other characteristics about the fixed point numbers that the programmer must be aware of. These are the number of bits to the left, and to the right, of the binary point. The primary concern in adding two fixed point numbers, as shown before, is that the binary points must be made to align. To accomplish this, the programmer must left or right shift one or both of the numbers. It is this shifting that must take into account both the number of magnitude and fractional bits in the numbers. Serious clipping errors can result by left shifting the high-order bits out of the left end of the AC or MQ; likewise truncation inaccuracies may accumulate by right shifting the low-order bits out of the right end of the AC or MQ. A clipping example is shown below from which an understanding of the truncation problem can be visualized also.



37.75 B16



3.125 B20

The programmer wants to add the two numbers, 37.75 B16 and 3.125 B20. If he were to write the following assembler program

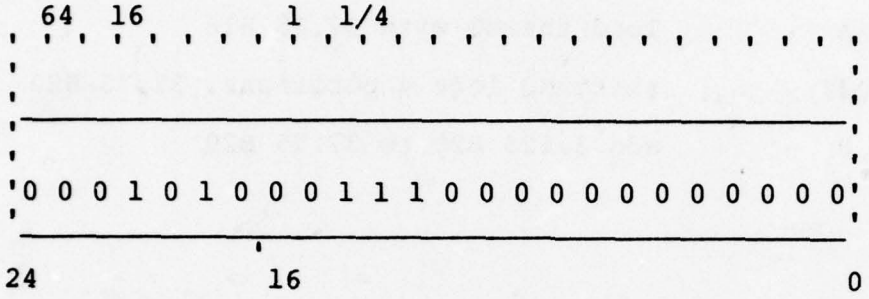
to perform the calculation, he would not get the desired result of 40.875.

```
          LDMQ A          load the MQ with 37.75 B16
          QLS  4          shift MQ left 4 positions, 37.75 B20
          ADMQ B          add 3.125 B20 to 37.75 B20
          *
          *
          *
          *
          *
A         DEC 37.75 B16
B         DEC 3.125 B20
```

The programmer failed to take into account that he clipped the 37.75 by left shifting some of the high-order bits out of the MQ in his attempt to align the binary points. The top two bits were lost; it was effectively 5.75 B20 that was added to 3.125 B20 to yield -7.125 B20 instead of the expected result. The program that should have been written to take into consideration the clipping possibility is shown below.

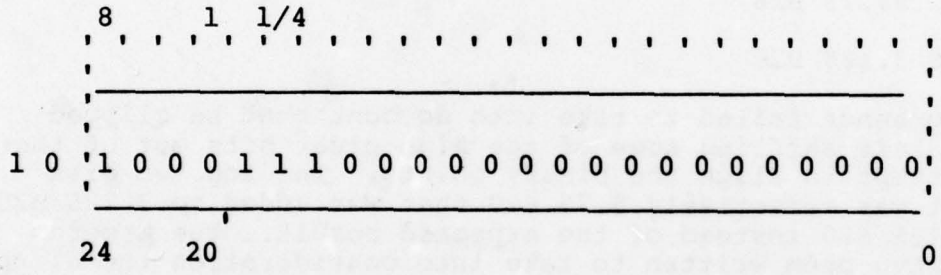
```
          LDMQ B          load the MQ with 3.125 B20
          AQRS 4          shift MQ right 4 positions, 3.125 B16
          ADMQ A          add 37.75 B16 to 3.125 B16
          *
          *
          *
```

The result of the correctly written program is 40.875 B16 and is shown below as it appears in the MQ.

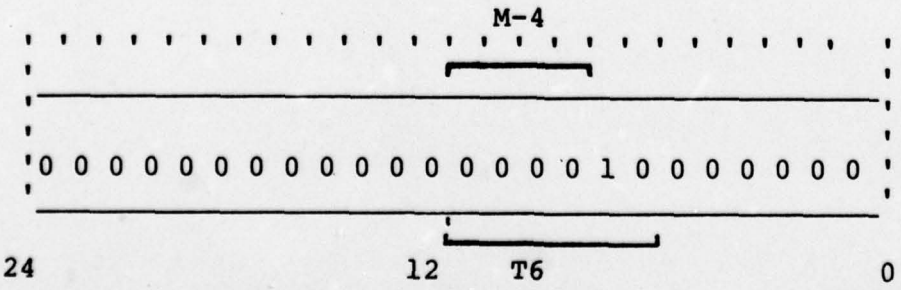


The result of the incorrectly written program was -7.125 B20 and is shown below as it appeared in the MQ. Note the high-order bits that were lost.

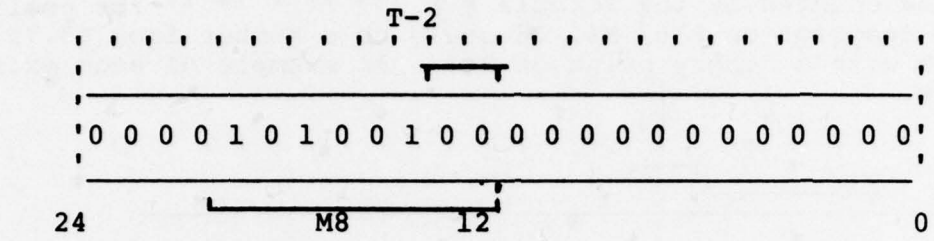
LOST



A convenient notation was developed for describing the binary point and bit content of fixed point numbers. The data number, A, above is said to be B16, M6, T2, where the B field is the position of the binary point, the M field is the number of magnitude bits, and the T field is the number of fraction bits. These definitions hold for positive M and T fields, but negative fields are also possible. Negative M is defined to be the number of bits to the right of the binary point before a fraction bit is reached, and negative T is the number of bits to the left of the binary point before a magnitude bit is reached. Both negative situations are illustrated below.

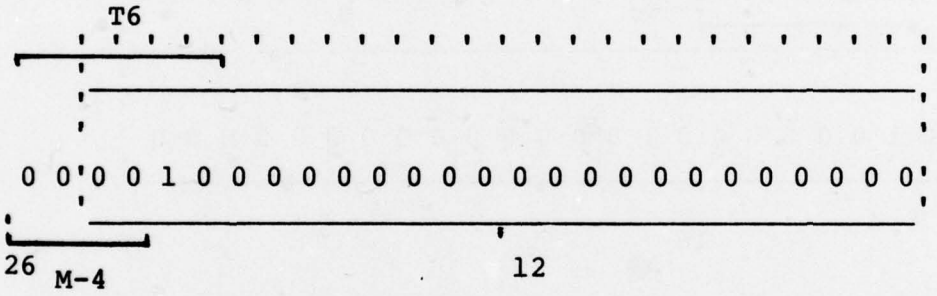


The number is .03125 B12 with M-4, T6.

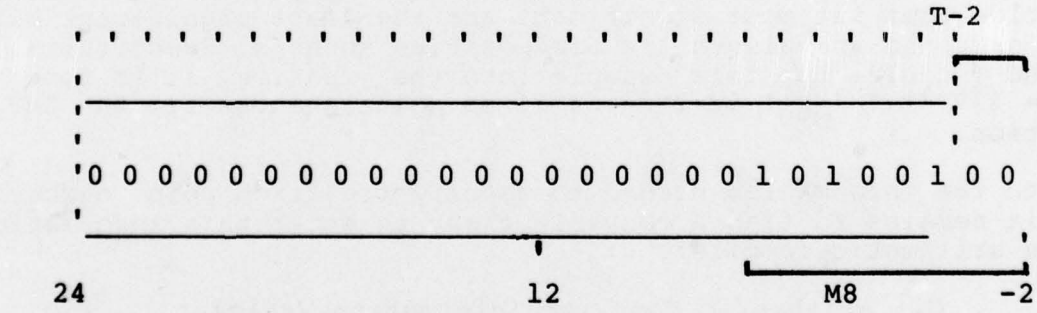


The number 164 B12 with M8, T-2.

The negative M and T fields are important because they can allow a number to be shifted such that the binary point can reside outside of the operational register without losing any information bits. In the first example above, the only information in the number lies in the bits between the end of the M field and the end of the T field. In the second example this is also true. Therefore the numbers can be shifted as shown below with no harm done.



The number is .03125 B26 with M-4, T6.

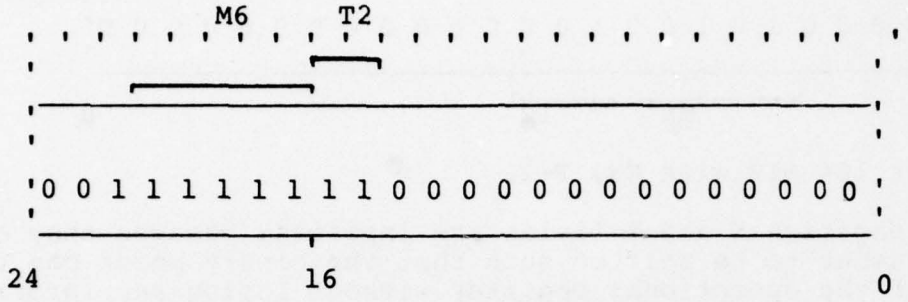


The number is 164 B-2 with M8, T-2.

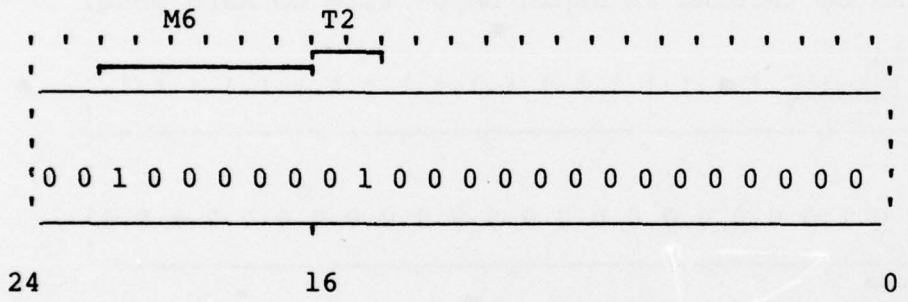
If the M and T fields are preserved before any mathematical operation, then no inaccuracies will result because of the operation other than the inherent quantatization errors that arise from representing numbers with only 24 bits.

It should be clear that the BMT representation does not uniquely define any fixed point number with the exception of some trivial

cases. In the general case, sets of numbers are specified whose elements can be counted by the formula # = $2^{(M-1) + (T-1)}$ for positive M and T. The description B16, M6, T2 could be a number from 63.75 to 32.25, both with a binary point of B16. An example of each extreme is shown below.



This number is 63.75 B16 with M6, T2.



This number is 32.25 B16 with M6, T2.

Notice that the most significant and the least significant bits of both extremes are always 1's for positive numbers. Substituting the M and T fields for this example into the equation, it is found that # = $2^{(6-1) + (2-1)}$ or 2^6 = 64 fixed point numbers fit the BMT description.

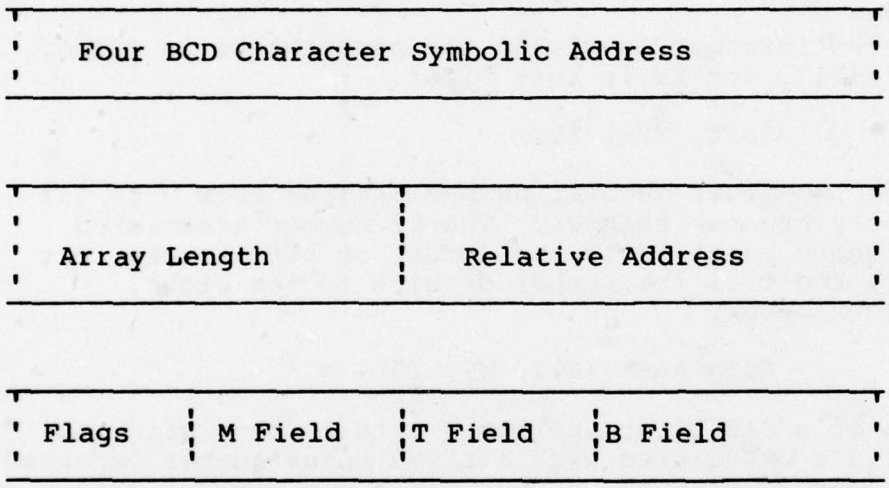
With the information needed to specify the fixed point numbers known, it remains to find a convenient way to enter this information into the arithmetic compiler.

C. Arithmetic Compiler Information Vehicles

Pass 1 forms the symbol table for the program being assembled. It contains a three-word set for each symbolic address that either appears in the label field of an instruction or is entered by a pseudo-op instruction. Reference 1 section 5.10 defines the pseudo-op to be a non-executable instruction that affects the assembly process or the program being assembled during that assembly process. Since the symbol table resides in core through all passes of DOPE, it is chosen to hold the BMT data and other descriptions for the

equation arguments which are also symbolic addresses. The limited capability of the symbol table pseudo-ops is expanded, to incorporate the old method of label entry and the BMT storage requirement, into three new pseudo-op instructions: ASSN, DIMN, and CORR. The ASSN statement stores and associates BMT information with one or more symbolic addresses, the DIMN statement enters the array length for those symbolic addresses that are single dimensioned variables, and the CORR statement gives the programmer the capability of affecting the BMT information within an equation expansion on the macro level. DIMN is evaluated during pass 1, while the ASSN and CORR evaluations occur in pass 1.5.

The format for the three-word set for every entry in the symbol table is shown below.



The SYMB word stores the four-character BCD mnemonic symbolic address, six bits per character. The SYML word has two fields. The first is reserved for the relative location of the label as it appears in the program, and the second holds the array length if the symbolic address is a dimensioned array. The BMT information is stored in the SIFW word requiring only 18 of the available 24 bits.

Using the ASSN or DIMN pseudo-ops eliminates the need of having symbolic addresses also appear as labels in the source program. The compiler will reserve locations at the end of the program for all single entry or dimensioned array symbolic addresses in the order that they appear in the pseudo-ops. If the programmer wishes to have the reserved locations other than at the end of the program, he must have the symbolic addresses appear as labels.

1. The ASSN Pseudo Operation

The function of the ASSN pseudo-op is to store the BMT parameters that are associated with an equation argument in the symbol table. The syntax of the card image is shown below.

```
Card Column
8     13                                                        72
ASSN XXXX,XXXX, ··XXXX(  )  XXXX, ··XXXX(  )  XXXX,  ···
```

XXXX is a maximum four-character argument. Enclosed within the set of parentheses is the BMT information which is associated with the preceding string of arguments. The second string of arguments is associated with the BMT parameters within the second set of parentheses. Argument strings and parentheses sets may continue through card column 72.

The format of the information within the parentheses is rigid. To 'not specify' a field, the YY is left blank.

(BYY, MYY, TYY)

YY is a one- or two-digit decimal number ranging from 0 to 63. Typically it is rarely greater than 24. The YY number associated with the B is the binary point, M is the number of bits to the left of the binary point, and T is the number of bits to the right. Two examples are shown below.

ASSN ALFA (B12, M9, T6)

The parameters of a B12 binary point, 9 bits to the left and 6 bits to the right,are associated with a fixed point number referred to by the symbolic address of ALFA.

ASSN A, B, GAMA (B6, M2, T3) C, BD (B18, M3, T17)

The parameters of (B6, M2, T3) are associated with the symbolic addresses A, B, and GAMA. The parameters of (B18, M3, T17) are associated with C and BD.

In the B field of the SIFW word, the two's complement form of the BYY field is stored. This is not the case with the MYY and TYY fields. It is convenient for the programmer to think in terms of number of bits to the left and right of the binary point. It is convenient for the compiler to think in terms of number of shifts that the number can be left or right shifted. The ASSN routine transforms both M and T from 'number of bits' to 'number of shifts' and stores these values in the SIFW word. The transformation equations are shown below.

```
#l.s. = 22-(B+M)      #l.s. is the number of left shifts

#r.s. = B-T           #r.s. is the number of right shifts
```

2. The DIMN Pseudo Operation

The function of the DIMN pseudo-op is to store the array length associated with a label in the SYML word. The syntax of the card image is shown below.

Card Column

8 13 72

DIMN XXXX,XXXX,··XXXX( ) XXXX,··XXXX( ) XXXX···

The only difference between the above and the syntax of the ASSN card image is the content of the set of parentheses; the decimal length of the array, a maximum of 2047, is the data enclosed. Two examples are shown below.

DIMN ALFA (1024)

The array referred to by the label ALFA will have a 1024-word block reserved at the end of the program if it does not also appear as a label.

DIMN A, B, GAMA (20) C, BD (128)

The arrays referred to by the symbolic addresses A, B, and GAMA will all have 20-word blocks reserved in the order of A first followed by B and GAMA at the end of the program. Two 128-word blocks immediately follow, C and BD respectively. These blocks will be reserved at the end of the program only if they do not also appear as labels.

3. The CORR Pseudo Operation

The function of the CORR pseudo-op is to affect the result of a normal compilation by overwriting the BMT description generated for the result of a macro operation. The programmer can use this to affect the code being generated if he knows more about his data than just the BMT information.

As in the ASSN and DIMN pseudo-ops, the CORR information appears in the variable field of the card image. The syntax is shown below.

Card Column

8 13 72

CORR $YY( ) $YY( ) $YY( ) · · · · $YY( )

The CORR card is placed anywhere before the equation it is to affect. The $YY is the delimiter on the macro image that the information within the set of parentheses refers to. The format

within the parentheses is identical to that of the ASSN pseudo-op also allowing one or more of the fields to be missing. The BMT information that does appear is placed on the symbol table associated with the $YY which is treated as a symbol. Each time the BMT parameters of a macro result are calculated, a scan of the symbol table is made with the delimiter that appears on the macro card image. If a match is found, then the associated BMT parameters overwrite the ones calculated by the compiler. The $YY symbol is then deleted from the symbol table. Two examples are shown below.

```
CORR $1(B12,M3,T2)
```

The result generated in the first macro of the following LET statement will be treated as a B12 number with three magnitude and two truncation bits. The BMT description generated by the compiler will be ignored.

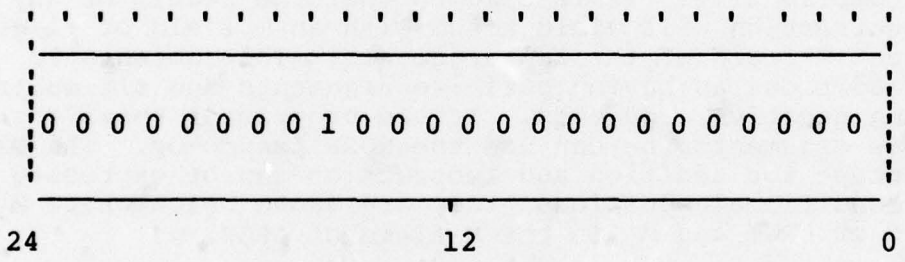
```
CORR $3(B,M3,T) $13(B,M4,T0)
```

The result of the third macro in the following LET statement will have the B and T descriptions generated by the compiler, but the M field will be M3. The thirteenth macro of the same LET statement will have the compiler-generated description of its B field, but will have magnitude and truncation descriptions of M4 and T0.

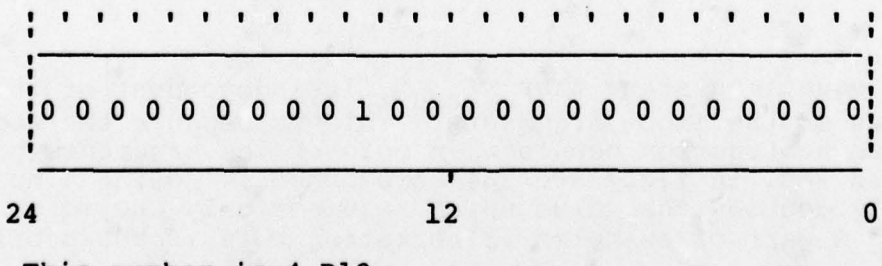
CHAPTER IV

DETERMINATION OF M AND T

An important feature of the arithmetic compiler is the determination of the BMT descriptions of the outputs of the macros, the intermediate results. They are derived from the BMT descriptions of the equation arguments given in the ASSN pseudo-ops. BMT parameters describe the maximum number of bits of the fixed point data to eliminate clipping and truncation errors. There is a problem with this technique that shows up in a long string of mathematical operations. The results of each macro have a tendency to become more and more 'safe' with respect to clipping and truncation. It is conceivable that a result will have its #l.s. and #r.s. parameters tend toward zero in the limit. When this happens, a result is characterized as having all 24 bits essential when in reality they may not be. A simple example of clipping that illustrates the problem is shown below.



This number is 8 B12.



This number is 4 B12.

At compile time, the arithmetic compiler only knows that B12, M4, T0 number is to be added to a B12, M3, T0 number. To avoid clipping, it has to assume that the maximum positive numbers fitting the descriptions are going to be added. It therefore assumes that 15 B12 and 7 B12 will produce 22 B12 which requires five magnitude

bits, B12, M5, TO. This assumption of maximum arguments has produced an extra bit of safety that is not necessary. The result of the addition at run time for the actual numbers 8 B12 and 4 B12 yields 12 B12 which can be represented with just four magnitude bits, B12, M4, TO.

The situation does exist that will cause a snowball effect of safety. The arithmetic compiler is designed to stop and produce an error message if safety chokes intermediate results to such an extent that they cannot be shifted, or would require greater than single precision mathematics to carry on the computation without error. The programmer is able to interpret the error message during pass 1.5 and insert a CORR pseudo-op in the source deck. When assembly is begun again, pass 1.5 will correct the $YY level of the equation expansion.

The calculation algorithms for the BMT parameters of each of the seven mathematical operations are given in the following discussions as they are implemented in Macro Optimization of pass 1.5.

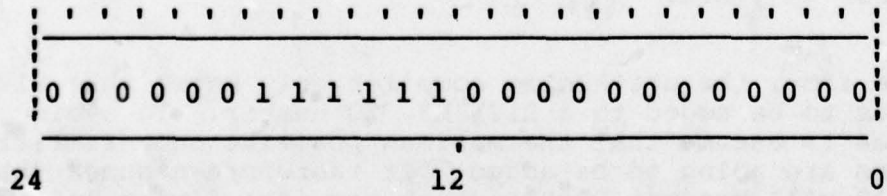
A. Addition and Subtraction M and T Fields

The calculation of #l.s. and #r.s. parameters for addition and subtraction are the same because the sign of the arguments is not known at compile time. It is assumed that the result of any addition or subtraction will yield a sum with an M field of +1 greater than the largest M field of the two arguments. This amounts to treating all additions as having positive arguments and all subtractions as having negative arguments. If the programmer knows the polarity of his arguments, he can use the CORR pseudo-op. The maximum number concept for addition and subtraction can be expressed as a pair of conditional equations; they are shown below where M_1 is the M field of ARG1 and M_2 is the M field of ARG2.

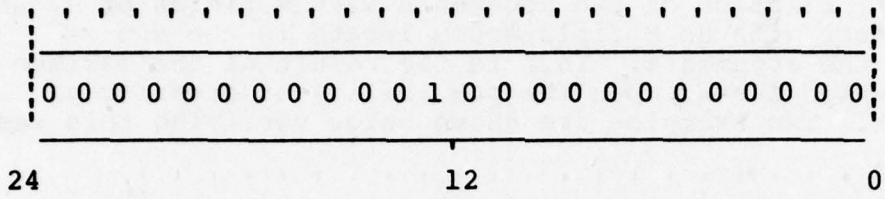
$$M_{result} = M_1 + 1 \qquad \text{if } M_1 \leq M_2$$

$$M_{result} = M_2 + 1 \qquad \text{if } M_1 < M_2$$

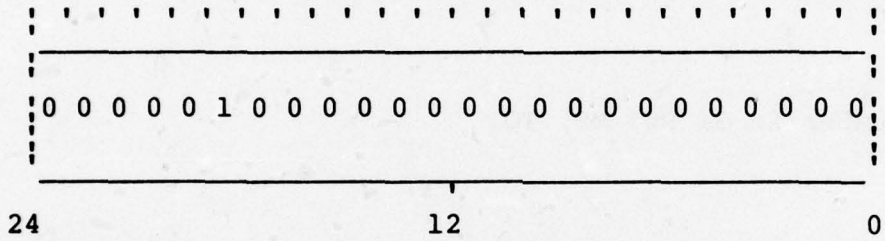
The above equations state that M_{result} is independent of the numerical values of the input arguments. This is because the worse case addition or subtraction occurs when both of the arguments have the same M field and, in fact, are the same number. Adding a number to itself merely doubles the value which requires only one more magnitude bit. A pair of examples illustrating this is shown below.



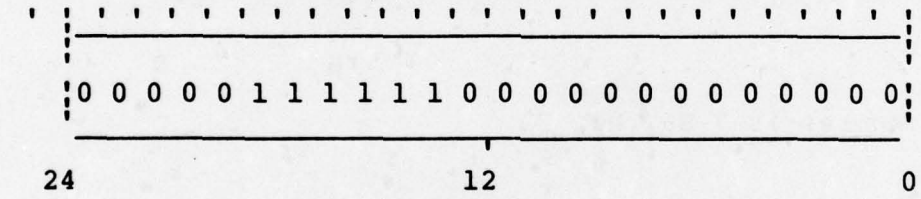
This number is 63 B12, M6, TO.



This number is 1 B12, M1, TO.



This number is the sum of 63 B12 and 1 B12 which is 64 B12, M7, TO. The M field of the sum is one greater than the largest M field of the input arguments.



This number is the sum of 63 B12 and 63 B12 which is 126 B12, M7, TO. The M field of this sum is also one greater than the largest M field of the input arguments.

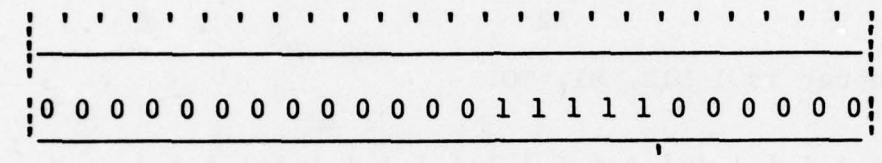
The choice for the T field of the result is the smallest field of the input arguments. The conditional equations are shown below.

$T_{result} = T_1$ if $T_1 \leq T_2$

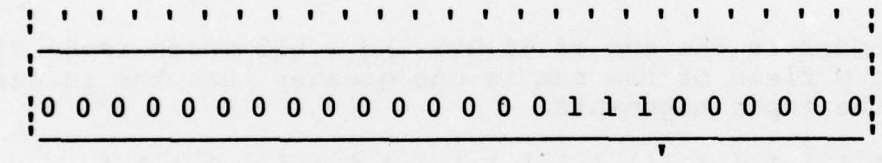
$T_{result} = T_2$ if $T_1 > T_2$

B. Multiplication M and T Fields

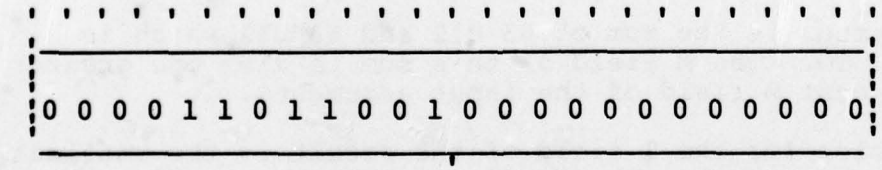
Binary multiplication of two arguments with M fields of M_1 and M_2 yields a product with an M field whose length is the sum of the M fields of the arguments. This is the result of the maximum number concept which leaves open the possibility of creating an extra safety bit. Two examples are shown below verifying this result.



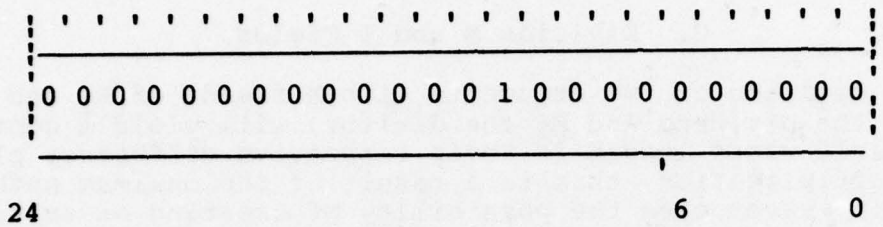
This number is 31 B6, M5, TO.



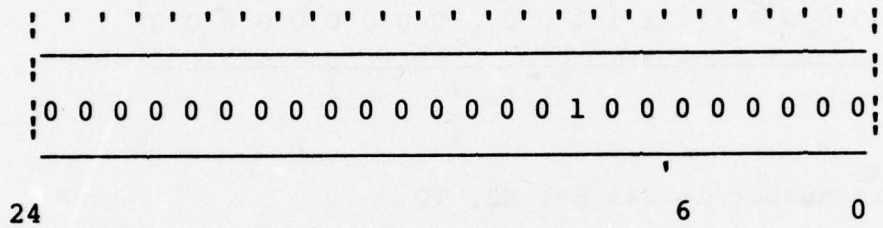
This number is 7 B6, M3, TO.



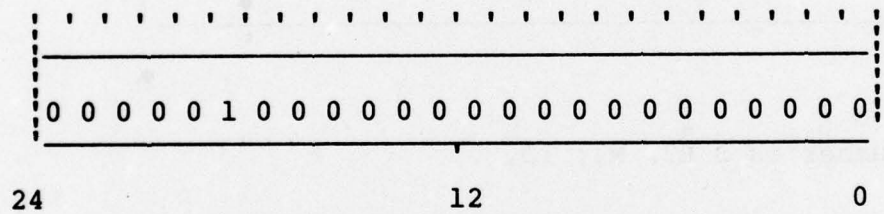
This number is the product of 31 B6 and 7 B6 which is 217 B12, M8, TO. The M field of the product is the sum of the M fields of the input arguments.



This number is 16 B6, M5, TO.



This number is 4 B6, M3, TO.



This number is the product of 16 B6 and 4 B6 which is 64 B12, M8, TO. The M field is calculated as the sum of the M fields of the input arguments which results in an extra bit of safety being generated.

The T field of the product is chosen to be the smaller field of the input arguments. The conditional equations are shown below.

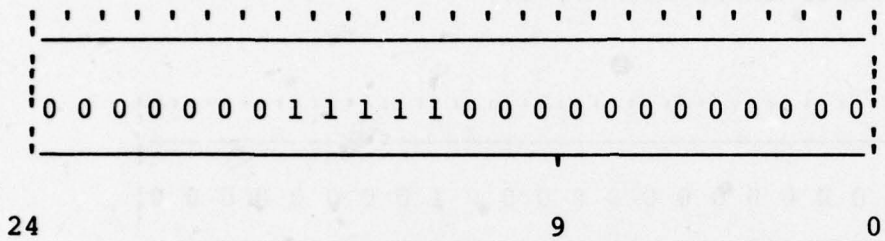
$$M_{result} = M_1 + M_2$$

$$T_{result} = T_1 \qquad \text{if } T_1 \leq T_2$$

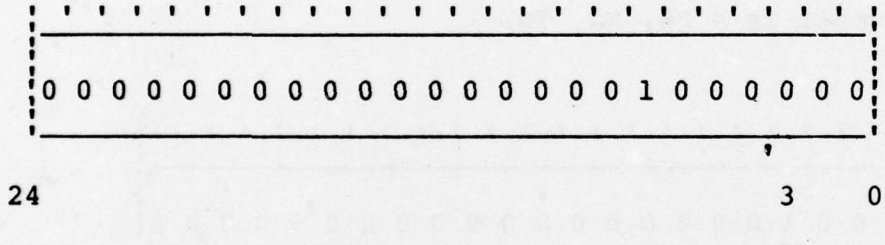
$$T_{result} = T_2 \qquad \text{if } T_1 > T_2$$

C. Division M and T Fields

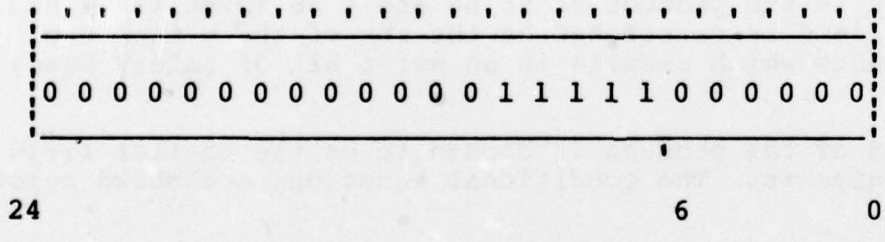
Binary division of two arguments with M fields of M_1 and M_2, where M_1 is the dividend and M_2 the divisor, will yield a quotient with an M field whose length is their respective difference plus 1. As in multiplication, this is a result of the maximum number concept which leaves open the possibility of creating an extra safety bit. Two examples are shown below illustrating this result.



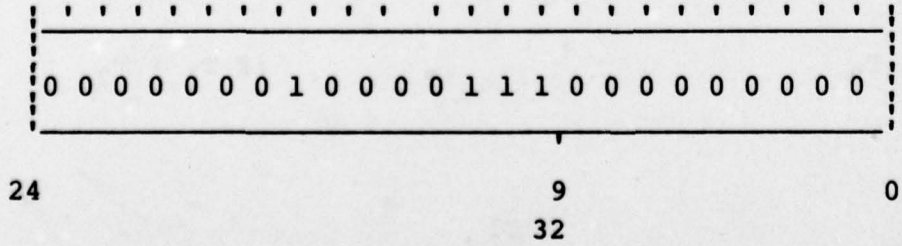
This number is 248 B9, M8, TO.



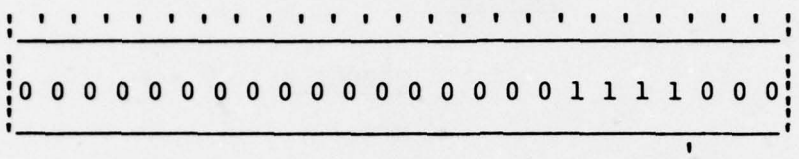
This number is 8 B3, M4, TO.



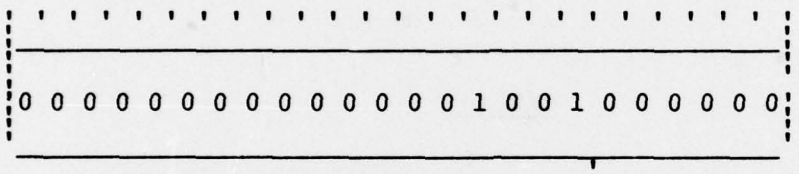
This number is the quotient of 248 B9 and 8 B3 which is 31 B6, M5, TO. The M field of the quotient is the difference of the M field of the dividend and the divisor + 1.



This number is 135 B9, M8, TO.



This number is 15 B3, M4, TO.



This number is the quotient of 135 B9 and 15 B3 which is 9 B3, M5, TO. The M field is calculated as the difference between the M field of the dividend and the divisor + 1 which results in an extra safety bit being generated.

The T field of the quotient is chosen to be the smaller field of the input arguments. The conditional equations are shown below.

$$M_{result} = M_1 - M_2 + 1$$

$$T_{result} = T_1 \qquad \text{if } T_1 \leq T_2$$

$$T_{result} = T_2 \qquad \text{if } T_1 > T_2$$

D. Subroutine Call M and T Fields

All subroutine calls used in equations assume that arguments are passed in the MQ with predetermined BMT fields. A table of allowable library subroutines and their BMT formats is kept within the compiler. If a programmer wishes to use his own subroutine call and enter the routine at run time, the compiler will assume it to have both input and output formats of B11, M11, T11. If the subroutine uses a different format, correction can be made with a CORR pseudo-op. The M and T ranges for the results of each subroutine as a function of the input are given below. The T field for the majority of the results is arbitrarily set to T6. If the

programmer is confident of the number of bits he wishes to carry, he can make use of the CORR pseudo-op again to enter his own T field.

LOG (e)

INPUT M	OUTPUT M	T = 6
11 = M ≥ 6	M = 3	
5 = M ≥ 3	M = 2	
2 = M ≥ 0	M = 1	
-1 = M ≥ -4	M = 2	
-5 = M ≥ -11	M = 3	

EXP (e)

INPUT M	OUTPUT M	T = 6
M = 3	M = 11	
M = 2	M = 6	
M = 1	M = 3	
M = 0	M = 2	

LOG2 (2)

INPUT M	OUTPUT M	T = 6
11 = M ≥ 9	M = 4	
8 = M ≥ 5	M = 3	
4 = M ≥ 2	M = 2	
1 = M ≥ 0	M = 1	
-1 = M ≥ -2	M = 2	
-3 = M ≥ -6	M = 3	
-7 = M ≥ -10	M = 4	

EXP2 (2)

INPUT M	OUTPUT M	T = 6
M = 4	M = 11	
M = 3	M = 8	
M = 2	M = 4	
M = 1	M = 2	
M = 0	M = 1	

LOGC (10)

INPUT M	OUTPUT M	T = 6
11 = M $\geq$ 6	M = 2	
5 = M $\geq$ 0	M = 1	
-1 = M $\geq$ -10	M = 2	

SIN(rad)

INPUT M	OUTPUT M	T = 6
M $\geq$ 0	M = 1	
M $\leq$ -1	M = M_{Input}	

COS (rad)

INPUT M	OUTPUT M	T = 6
any M	M = 1	

SIND(deg)

INPUT M	OUTPUT M T = 6
$M \geq 7$	M = 1
$-6 \leq M \leq 6$	$M = M_{Input} - 5$

COSD(deg)

INPUT M	OUTPUT M T = 6
any M	M = 1

ASIN

INPUT M	OUTPUT M T = 6
$M \leq -1$	$M = M_{Input}$
M = 0	M = 1

ACOS

INPUT M	OUTPUT M T = 6
any M	M = 1

ASND

INPUT M	OUTPUT M T = 6
$M \leq -1$	$M = M_{Input} + 5$
M = 0	M = 7

ACSD

INPUT M	OUTPUT M	T = 6
any M	M = 7	

SORT

INPUT M	OUTPUT M	T = T_{In}
any M	M = $M_{\frac{Input}{2}}$	

ATAN

INPUT M	OUTPUT M	T = 6
any M	M = 1	

ATND

INPUT M	OUTPUT M	T = 6
any M	M = 7	

E. Equality M and T Fields

The equality operation is always the last operation in the evaluation of an equation. If the argument on the left of the equal sign does not appear in an ASSN statement or has all three BMT fields unspecified, then the BMT fields of the final result on the right side of the equation will be assigned to it. If the argument is defined by an ASSN with only the B field specified, then the result on the right is shifted to make it obey the ASSN B field. The M and T fields of the right side result are assigned to the argument. In summary, the BMT fields of an argument that are specified in an ASSN statement overwrite the ones generated by the right side

of the equation . When B is not specified, the argument is shifted to make its #1.s. parameter equal to zero.

F. Exponentiation M and T Fields

The exponentiation operation uses two subroutines, EXP2 and LOG2, to realize the raising of an argument to an argument power. A multiplication is also performed within the macro in the order LOG2, multiply, and EXP2. Using the rule for multiplication and the transformations for the subroutines, the M and T fields for the result of exponentiation can be obtained.

CHAPTER V

SPECIFIC MACRO REALIZATION

The Macro Approach discussion of section II states that the compiler will create a string of specific macros to most efficiently realize the equation. For the macro approach to be feasible, all the specific macro expansions must fit in the remaining pass 2 core after the Macro Expansion routine itself has been entered; there are only 920 decimal locations to house both the routine and the expansions.

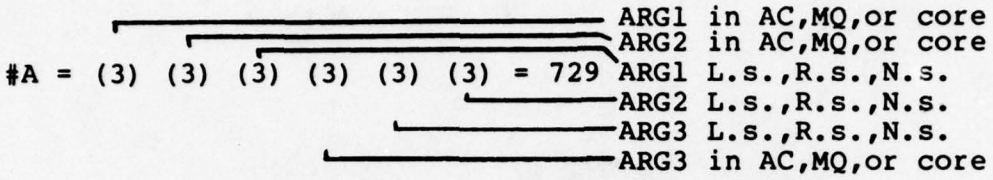
The number of the specific macros for each operation is based on the problem of having an argument with three arbitrary parameters, #l.s., #r.s., and B, being operated on by another argument also with three arbitrary parameters. Prior to the operation, each argument can be in core, the AC, or the MQ. At the completion of the operation, the result can be stored in core, left in the MQ, or left in the AC with an arbitrary binary point different from the one immediately generated by the operation.

The addition and subtraction macros were investigated first since they require alignment of the binary points which is not necessary for the other operations. Principles and short cuts revealed in these investigations are applied to the other operations.

A. Addition and Subtraction Investigation

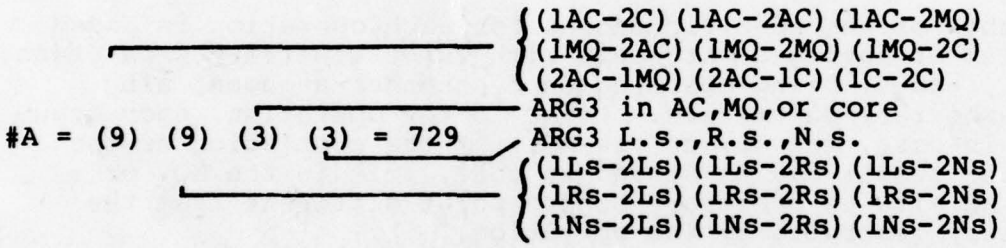
Section IVA stated that there is no difference between addition and subtraction in the calculation of the MT data. The general macro mnemonic, ADDM, is the name of the set that contains all the addition specific macros; each specific macro is an element of ADDM. SUBM is the name of the set that contains all the subtraction specific macro elements; they have a one-to-one correspondence with each element of ADDM. In the discussions to follow, addition will be synonomous with both addition and subtraction. This investigation will determine the minimum number of specific macros necessary to realize any possible addition situation.

The minimum number of elements in ADDM, and the form of each, is found by using the product rule of combination and permutation theory. First the maximum theoretical number of elements is found based on the arbitrary parameters of the arguments, then the elements that are meaningless or not unique are eliminated. In the product rule shown below, the symbol #A stands for the number of add macros. CORR flexibility is not taken into account in the element elimination process.



With an average length of five instructions per macro, there are (5)(729) = 3645 memory locations needed to hold the ADDM set alone. Only (920 - Macro Expansion code) are available for all the operations, so element elimination is critical.

Combining the location of the input argument and the possible shift, the product rule can be rearranged. For brevity, 1 and 2 will refer to ARG1 and ARG2 respectively. As an example, 1AC-2C will be ARG1 in the AC and ARG2 in core.



Only one of the input arguments can be in the AC or MQ prior to macro entry even though addition can occur between operational registers. This is consistent with optimization between macros, and not across, as stated in section IA. It is also impossible for both ARG1 and ARG2 to be simultaneously in the AC, or the MQ. Therefore the following combinations of input arguments can be eliminated.

ARG1 in AC, ARG2 in AC	(1AC-2AC)
ARG1 in MQ, ARG2 in MQ	(1MQ-2MQ)
ARG1 in AC, ARG2 in MQ	(1AC-2MQ)
ARG1 in MQ, ARG2 in AC	(1MQ-2AC)

This is a significant reduction in the number of elements in ADDM.

#A = (5)(9)(3)(3) = 405

It is not obvious, but within the remaining 405 elements there is an extra degree of freedom that has no effect on the result of the addition as it is described by its MT parameters. It is immaterial where within the 24 bits that the addition is performed as long as the binary points of the input arguments align and the #1.s.

and #r.s. barriers are not violated. Since it does not matter, a restriction can be imposed without any effect on the possible addition situations that can be realized by ADDM.

There are a number of possible restrictions, but the most advantageous in both the minimization of the number of specific macros and in computational accuracy is the #1.s. parameter zero criterion. It requires the shifting of the input arguments such that the result has its #1.s. parameter equal to zero. In the addition of two arguments, the one with the largest M field is left shifted until its #1.s. parameter is zero. The other argument is shifted to align the binary points. The immediate result of the addition has an M field of +1 greater than the largest M field of the input which yields a #1.s. parameter of -1. The result is right shifted one position to conform to the criterion. Computational accuracy is enhanced by the left shifting because an unofficial T field of arguments and results is carried throughout the macro string that can compensate for the choice of smallest T field for results and T6 assumption for subroutine calls. Below is shown a further detailed account of the remaining 405 specific macros according to the location of the input arguments.

1C-2MQ	(9)	(3)	(3) = 81	ARG1 in core, ARG2 in MQ
1C-2AC	(9)	(3)	(3) = 81	ARG1 in core, ARG2 in AC
2C-1MQ	(9)	(3)	(3) = 81	ARG2 in core, ARG1 in MQ
2C-1AC	(9)	(3)	(3) = 81	ARG2 in core, ARG1 in AC
1C-2C	(9)	(3)	(3) = 81	ARG1 in core, ARG2 in core
			405	add macros

With every macro always right shifting the result one position to meet the criterion, the no shift and left shift possibilities of the result are eliminated. The criterion also eliminates the possibility of shifting both input arguments right. It is impossible to shift both right and obtain a result that has to be shifted right one position to make its #1.s. parameter equal to zero. The elimination of the right shift of both input arguments reduces the input shift combination from (9) to (8). The elimination of the no shift (Ns) and left shift (Ls) of the result reduces the output shift combination from (3) to (1).

		Eliminated Combinations
1C-2MQ	(8)(1)(3) = 24	(1Rs-2Rs) ARG3 Ls, Ns
1C-2AC	(8)(1)(3) = 24	(1Rs-2Rs) ARG3 Ls, Ns
2C-1MQ	(8)(1)(3) = 24	(1Rs-2Rs) ARG3 Ls, Ns
2C-1AC	(8)(1)(3) = 24	(1Rs-2Rs) ARG3 Ls, Ns
1C-2C	(8)(1)(3) = 24	(1Rs-2Rs) ARG3 Ls, Ns

$\overline{120}$ add macros

The hardware limitation discussed in section IB restricted arithmetic right shifting to the MQ alone. The AC can perform only logical shifts; it cannot execute an arithmetic right shift which is necessary with negative arguments. Below are shown the macro possibilities that this fact eliminates.

		Eliminated Combinations
1C-2MQ	(8)(1)(3) = 24	
1C-2AC	(6)(1)(3) = 18	(1Ls-2Rs)(1Ns-2Rs)
2C-1MQ	(8)(1)(3) = 24	
2C-1AC	(6)(1)(3) = 18	(2Ls-1Rs)(2Ns-1Rs)
1C-2C	(8)(1)(3) = 24	

$\overline{108}$ add macros

The #1.s. parameter zero criterion assures that all results that are generated by one of the macro sets will have their #1.s. parameter equal to zero. The only way a macro can have an input argument appear in an operational register is for it to have been left there by a previous macro. Therefore by definition it has its #1.s. parameter equal to zero and cannot be left shifted. Below are shown the macro possibilities that this fact eliminates.

		Eliminated Combinations
1C-2MQ	(5)(1)(3) = 15	(2Ls-1Ls)(2Ls-1Rs)(2Ls-1Ns)
1C-2AC	(3)(1)(3) = 9	(2Ls-1Ls)(2Ls-1Rs)(2Ls-1Ns)
2C-1MQ	(5)(1)(3) = 15	(2Ls-1Ls)(2Ls-1Rs)(2Ls-1Ns)
2C-1AC	(3)(1)(3) = 9	(2Ls-1Ls)(2Ls-1RS)(2Ls-1Ns)
1C-2C	(8)(1)(3) = 24	

$\overline{72}$ add macros

Before any further reductions are made, the table is shown again with the remaining shift possibilities innumerated for clarity.

1C-2MQ	(1Ls-2Rs) (1Ls-2Ns) (1Rs-2Ns) (1Ns-2Rs) (1Ns-2Ns)
1C-2AC	(1Rs-2Ns) (1Ls-2Ns) (1Ns-2Ns)
2C-1MQ	(2Ls-1Rs) (2Ls-1Ns) (2Rs-1Ns) (2Ns-1Rs) (2Ns-1Ns)
2C-1AC	(2Rs-1Ns) (2Ls-1Ns) (2Ns-1Ns)
1C-2C	(1Ls-2Ls) (1Ls-2Rs) (1Ls-2Ns) (1Rs-2Ls) (1Rs-2Ns)
	(1Ns-2Ls) (1Ns-2Rs) (1Ns-2Ns)

In the 1C-2MQ line, ARG2 is in the MQ so ARG1 cannot be right shifted. This eliminates the (1Rs-2Ns) possibility.

The only difference between the (1Ls-2Ns) and the (1Ns-2Ns) possibilities of 1C-2MQ and the same ones of 1C-2AC is the operational register that ARG2 is located in. The redundancy is eliminated by incorporating both possibilities into 2C-1MQ. A similar situation exists between the (2Ls-1Ns) and the (2Ns-1Ns) of 2C-1MQ and the same ones of 2C-1AC. Both possibilities are incorporated into 2C-1MQ. The table below shows the specific macro elements that will be contained in ADDM; the minimum set has been found.

1C-2MQ	(4) (1) (3) = 12	(1Ls-2Rs) (1Ls-2Ns) (1Ns-2Rs) (1Ns-2Ns)
1C-2AC	(1) (1) (3) = 3	(1Rs-2Ns)
2C-1MQ	(4) (1) (3) = 12	(2Ls-1Rs) (2Ls-1Ns) (2Ns-1Rs) (2Ms-1Ns)
2C-1AC	(1) (1) (3) = 3	(2Rs-1Ns)
1C-2C	(8) (1) (3) = 24	all eight possibilities
	$\overline{54}$	add macros in ADDM

The #1.s. parameter zero criterion has been instrumental in reducing ADDM to just 54 specific macro elements. SUBM also has 54 elements. The eight subsets that house the elements are shown in Appendix C complete with the expansion code of each element. They are AX1X through AX8X for addition, SX1X through SX8X for subtraction; one subset for each possible shift combination.

The technique for determining the shift parameters of the arguments and binary point of the sum can be expressed as conditional equations.

$$PAR1 = \begin{cases} \#1.s. \text{ of ARG1 if } M_1 > M_2. \\ 0 \text{ if } B_1 = B_2 \text{ and \#1.s. of ARG1 or ARG2 is zero.} \\ \text{difference between the adjusted binary point of ARG2 and the unadjusted binary point of ARG1 if } M_2 > M_1. \end{cases}$$

$$PAR2 = \begin{cases} \#1.s. \text{ of ARG2 if } M_2 > M_1. \\ 0 \text{ if } B_2 = B_1 \text{ and \#1.s. of ARG2 or ARG1 is zero.} \\ \text{difference between the adjusted binary point of ARG1 and the unadjusted binary point of ARG2 if } M_1 > M_2. \end{cases}$$

$$PAR3 = 1$$

$$B_{ARG3} = \begin{cases} \text{adjusted binary point of ARG1} - 1 \text{ if } M_1 \geq M_2. \\ \text{adjusted binary point of ARG2} - 1 \text{ if } M_2 > M_1. \end{cases}$$

Adjusted binary point is defined to be the sum of the #1.s. parameter of an argument and its binary point. Adjusted binary points are compared to determine the PAR1 and PAR2 of a macro to be consistent with the #1.s. parameter zero criterion.

B. Multiplication Investigation

The variety of possible multiply situations is not as great as in addition and subtraction because alignment of the binary points is not necessary. The #1.s. parameter zero criterion also simplifies the macro investigation. The individual multiply macro must however take into account the limitations imposed by the DISAC hardware. Only positive numbers can be multiplied with the most significant bits of the double precision result in the AC and the least significant in the MQ. A further restriction dictates that one of the arguments must be in memory and the other in the MQ prior to the execution of the operation. The input arguments are shifted such that the result will have its #1.s. parameter equal to zero and will appear in the AC.

The mechanics of fixed point multiplication can be expressed in terms of the binary points of the input arguments even though the operation itself is independent of the concept. Two arguments with binary points of BX and BY respectively, will yield a result with a binary point of BX + BY. As an example, if ARG1 is B18 in the MQ with the AC zero, and ARG2 is B12 in core, the double precision product will have a B30 binary point treating the AC and the MQ as a single 48-bit register. This was first introduced in section IVB without explanation and without the restrictions of the #1.s. parameter zero criterion imposed.

The restriction of a single precision result using only positive arguments is realized by left shifting the input arguments so the result will have the greatest binary point possible. This insures that the M field of the result will be left justified in the AC

(most significant part) and the T field will extend over into the MQ (least significant part). By retaining the AC as the single precision result, the most significant parts of the M and T fields will describe the result. A B30 binary point becomes (30 - 24) = 6 or B6 with the discarding of the MQ bits. This single precision restriction independently approaches the design of the #1.s. parameter zero criterion though not completely. There still exists some latitude as to what extent the arguments are left shifted. If both arguments are shifted such that their #1.s. parameters are zero, the immediate result of the multiplication would have a #1.s. parameter equal to 2. The result would have to be left shifted twice to make it conform with the criterion. The left shift of the result is cumbersome since addition and subtraction always right shift the result one position. Since the input arguments are made positive, and DISAC's hardware is not a signed multiplier, they can be shifted two positions beyond the #1.s. parameter barrier without clipping the data. If one of the input arguments was left shifted until it had a #1.s. parameter equal to -2 and the other shifted until its #1.s. parameter was -1, then the multiplication would yield a result that would have to be right shifted one position to make the #1.s. parameter equal to zero. This is desirable because now multiplication, addition, and subtraction would all shift results in the same fashion, but this is an artificial shift. The shift instruction can be saved by shifting each input argument until the #1.s. parameter is equal to -1. The result of the multiplication will always have its #1.s. parameter equal to zero with no additional shift required. The same effect could be achieved by the method of shifting one argument until its #1.s. parameter was -2 and the other until its #1.s. parameter was 0. This was considered but not chosen because it would at least triple the number of multiplication specific macros for only a 6% savings in code per macro and a 2% savings in speed. As a result, the general macro MULM has only one subset, MX1X, which contains the nine specific macro elements. The need for the nine elements can be determined from the product rule as in addition.

$$\#M = (3)(1)(1)(3) = 9$$

- (3) — ARG3 in AC, MQ, or core
- (1) — (1C-2C)(1MQ-2C)(2MQ-1C)
- (1) — (1Ls-2Ls)
- (3) — ARG3 Ns

The expanded list of each specific macro is shown in Appendix C.

The technique for determining the shift parameters of the arguments and the binary point of the product is shown below.

PAR1 = #1.s. of ARG1 + 1

PAR2 = #1.s. of ARG2 + 1

PAR3 = 0

$$B_{ARG3} = (B + \#1.s.)_{ARG1} + (B + \#1.s.)_{ARG2} - 22$$

C. Division Investigation

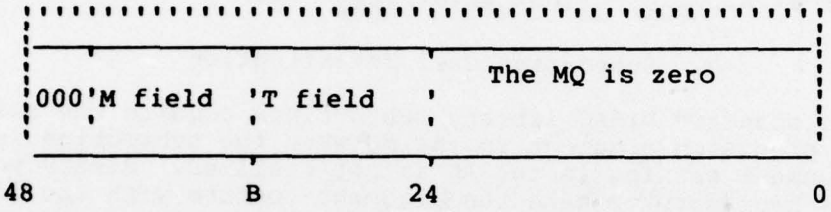
As with multiplication, division is not dependent on the alignment of the binary points of the arguments. The individual division macro must take into account certain restrictions imposed by the DISAC hardware. Only a positive number can be divided. A further restriction is that the magnitude of the most significant bits of the dividend must be less than the magnitude of the divisor. The divisor must also be a positive number and must be located in core. The single precision result must have its #l.s. parameter equal to zero and will appear in the MQ.

The mechanics of fixed point division can be expressed in terms of the binary points of the input arguments. If the dividend has a binary point of BX and the divisor has BY, then the quotient will have a binary point of BX - BY. As an example, if ARG1 (the dividend) is B18 in the MQ with the AC zero and ARG2 is B12 in core, the quotient will be B6 in the MQ. If the above binary points were the left justified versions of the arguments, then ARG1 has an M field of 4 and ARG2 had an M field of 10. The equations of section IVC dictate that the quotient should have an M field of (4 - 10 + 1) or -5. The -5 M field combined with the B6 binary point yields a #l.s. parameter of the quotient of 21. Therefore it would have to be left shifted 21 times to comply with #l.s. parameter zero criterion. This unacceptable development can be corrected by considering the dividend to be a double precision number.

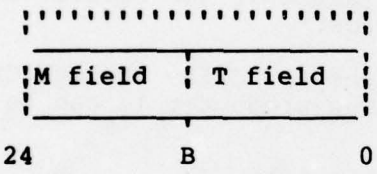
Multiplication of a single precision number in the MQ by a single precision number from core yields a double precision result in the AC and the MQ with the most significant bits in the AC. Since division is the inverse of multiplication, a double precision dividend with the most significant bits in the AC and least significant bits in the MQ yields a single precision quotient in the MQ when divided by a single precision number from core.

In the example above, ARG1 is placed in the AC to give the dividend a binary point of (18 + 24) or B42; its low-order bits in the MQ are set to zero. This double precision dividend is divided by ARG2 which is in core and has its binary point at B12. The M fields of the arguments are the same as before. After the division, the quotient is (42 - 12) or B30 in the MQ with an M field of -5. The #l.s. parameter of the quotient is -3 which would require a right shift of three to make it zero. The right shift is a possibility, but it can be avoided. The division was done with both ARG1 and ARG2 left justified with their #l.s. parameters equal to zero. Since only positive division is allowed, the divisor can be made to have its #l.s. parameter set to -2 before the division without clipping errors. ARG2 would now be B14, in the example, to yield a quotient of B27 if the dividend is shifted left 23 times from the MQ into the AC to make it have a #l.s. parameter of 1. Therefore the quotient is B27, (41 - 14), with an M field of -5 and a #l.s. parameter of zero which is consistent with the criterion.

With the placing of the ARG1 in the AC, the restriction of the magnitude of ARG2 (divisor) being greater than the magnitude of ARG1 becomes potentially hazardous. If this condition is not met, an erroneous quotient is generated by the division hardware. Below is illustrated how the -2 divisor and +1 dividend #1.s. parameter method tends to reduce the chances of an illegal division. For an error to occur, the dividend must have its M field exactly describe the data and the M field of the divisor must be over-specified by four bit positions.



The above is the representation of the double precision dividend with a #1.s. parameter equal to 1.



The above is the representation of the divisor with a #1.s. parameter of -2.

The first three bits of the dividend are always zero, and the divisor always has its M field left justified. This three-bit difference should afford enough safety to eliminate the division hazard for all practical purposes. Should it arise, it can be corrected with the CORR *pseudo-op*.

The criterion and the hardware limitations of DISAC cause DVSM to have only one subset, DX1X, to house all nine of the division specific macros. The need for the nine can be determined from the product rule.

$$\#D = (3)(1)(1)(3) = 9$$

(3) ARG3 in AC, MQ, or core
(1) (1C-2C)(1MQ-2C)(2MQ-1C)
(1) (1Ls-2Ls)
(3) ARG3 Ns

The expanded list of each specific macro is shown in Appendix C.

The technique for determining the shift parameters of the arguments and the binary point of the quotient is shown below.

PAR1 = #l.s. of ARG1 + 23

PAR2 = #l.s. of ARG2 + 2

PAR3 = 0

$$B_{ARG3} = (B + \#l.s.)_{ARG1} - ((B + \#l.s.)_{ARG2} - 21)$$

D. Subroutine Call Investigation

All the standard DISAC library subroutines require the passing of a single precision argument in the MQ when the subroutine is called. Argument passing in the AC is not realized. Binary point alignment is necessary to make the argument conform with the input format of the routine. The #l.s. parameter zero criterion states that the result of the macro must have its #l.s. parameter equal to zero. The immediate result of the subroutine must be shifted to comply before the result is stored in core or left in an operational register. For both input and output, all three shift possibilities exist, but only the right shift of the result is used to determine the number of specific macros.

The general macro CALM has three subsets, CX1X through CX3X, with a total of 15 specific elements. The product rule can be used to obtain this result.

#C = (1)(3)(3)(1) = 9
- ARG2 in core
- ARG2 Ls, Rs, Ns
- ARG3 Rs
- ARG3 in AC, MQ, or core

#C = (1)(2)(3)(1) = 6
- ARG2 in MQ
- ARG2 Rs, Ns
- ARG3 Rs
- ARG3 in AC, MQ, or core

15 call macros

Note that the conditions of the input argument being found in the AC and the result being left shifted are absent. This is because these conditions are not consistent with the criterion. The expanded list of each specific macro is shown in Appendix C.

The technique for determining the shift parameter for the input argument is to form the difference between the binary point of the argument and the binary point that the subroutine expects to see. The absolute value of this difference is the value of the parameter.

$$PAR2 = \left| B_{\text{format in}} - B_{ARG2} \right|$$

The shift parameter for the result is shown below.

$$PAR3 = (B_{\text{format out}} + M_{\text{format out}}) - 22$$

The table of input and output BMT formats for the library subroutines is given in section IVD.

E. Equality Investigation

The equality macro stores the result of a macro string in core. The operation is dependent on the binary points of the two arguments. The input, ARG1, must be shifted to meet the binary point of ARG2 if a binary point has been specified by an ASSN pseudo-op. All three shift possibilities exist for this operation. EQUM has three subsets, QX1X through QX3X, which contain only five specific macro elements in total. The product rule can again be used to show how the five were determined.

#E = (1)(3)(1) = 3
- ARG1 in core
- ARG1 Ls, Rs, Ns
- ARG2 in core

#E = (1)(2)(1) = 2
- ARG1 *in MQ*
- ARG1 Ls, Rs
- ARG2 in core

5 equality macros

The absence of leaving the result in the AC or MQ exists because the function of the macro is to store the result in core. The no shift possibility is missing when the input is found in the MQ because if no shift were to be performed, the previous macro would have stored the result in core itself and eliminated the need for the EQUM macro.

The technique for determining the shift parameter for the argument and the binary point of the result can be expressed as conditional equations.

$B_{ARG2} = B_{ARG1}$ if B_{ARG2} is not specified

$B_{ARG2} = B_{ARG2}$ if B_{ARG2} is specified

$PAR1 = B_{ARG2} - B_{ARG1}$ if B_{ARG2} is specified

$PAR1 = 0$ if B_{ARG2} is not specified

F. Exponentiation Investigation

Exponentiation is realized with the EXP2 and LOG2 subroutines, and a multiplication as stated in section IVF. This is possible because the raising of a variable to a variable power can be broken down into a simpler expression as shown below.

$$Y = a^b \qquad \text{or} \qquad ARG3 = ARG1^{ARG2}$$

$$\ln_2(ARG3) = \ln_2(ARG1^{ARG2})$$

$$\ln_2(ARG3) = (ARG2)\ln_2(ARG1)$$

$$2^{\ln_2(ARG3)} = 2^{(ARG2)\ln_2(ARG1)}$$

$$2^{\ln_2(ARG3)} = ARG3 = 2^{(ARG2)n_2(ARG1)}$$

The exponentiation of $ARG1^{ARG2}$ is realized by taking the logarithm to the base 2 of ARG1, multiplying it by ARG2, and finally the base 2 is raised to this product power. The calling of subroutines and multiplication have previously been defined as macros. It is possible to use them to realize exponentiation and thereby eliminate the need for any further investigation into creating a separate macro for exponentiation. Instead of seven mathematical operations being realized, only six need be considered from this point on. Shown below is the macro realization of EXPM.

```
EXPM B     A     $V00 $     CALM EXP2 B     $V00
                            MULM $V00 A     $V00
                            CALM LOG2 $V00  $V00
```

Being able to make this transformation is a considerable savings in pass 1.5 and pass 2. No evaluation routines are needed for exponentiation in pass 1.5 and the expansion code for all the exponentiation specific macros can be eliminated from pass 2.

CHAPTER VI

THE DOPE ASSEMBLER AND THE ARITHMETIC COMPILER

A minimum level, or two pass, assembler requires two scans through the source deck. The first pass enters into the symbol table every symbolic address that appears as a label on a source card along with its relative location within the program. The second pass recognizes the mnemonic in the instruction field and obtains a matching machine code from a stored table. If the instruction references a symbolic address in the program, a match is sought in the symbol table. The relative location is obtained and inserted into the 24 bit machine instruction. The N mnemonic instructions in the source deck have a one-to-one correspondence with the N machine code instructions in the object deck.

The idea of the minimum assembler is the foundation for the more sophisticated assemblers even as they approach the level of the compiler. The design philosophy behind the addition of compiler statements is to perform additional evaluations in pass 1 to transform the statements to assembler code. Pass 2 is then kept as that of the minimum assembler. With unlimited core, this can be done; but DISAC has a very stringent core restriction that causes distortion between the functional passes and the actual physical passes of the DOPE assembler. The addition of the arithmetic compiler compounds the problem to the extent that another physical pass has to be added, pass 1.5.

The Parsing Algorithm is the prior evaluation routine added to physical pass 1, Macro Optimization is the prior evaluation routine spilled over into pass 1.5, and Macro Expansion is a combination functional pass 1 - pass 2 routine residing in physical pass 2.

A. Pass 1 and the Parsing Algorithm

Three features require special emphasis within this pass because of their relation to the arithmetic compiler.

Subroutine calls to external routines require the creation of a transfer vector. This is explained in reference 1 section 5.15.3. In brief, the BCD name of each external routine is placed in a core location at the physical end of the program. When the System links the routines at run time, the absolute address of each subroutine replaces the BCD name. Access is made through these addresses, the transfer vector. Every different subroutine mnemonic that appears in a LET equation must also have an entry in the transfer vector.

The Parsing Algorithm keeps a table of such entries that is saved and interrogated when the transfer vector is created at the termination of pass 1.5.

The second feature is the incrementing of the location counter by some value if a general or specific macro is realized. For the specific macro, the exact length is obtained from a table, and the location counter incremented by that amount. This gap is filled in with the machine instructions that realize the specific macro by Macro Expansion of pass 2. It is the length of the longest macro of the set that is used to increment the location counter for the general macro. When Macro Optimization of pass 1.5 replaces the general macro mnemonic with the specific macro, the gap is appropriately adjusted.

The general macro decision is the third feature added because of the arithmetic compiler. The System always assumes that there are no general macros or LET statements in a program; therefore it will just load pass 1 and pass 2 in the assembly process unless directed otherwise. Upon recognition or generation of a general macro, a flag is set in the System selecting pass 1.5 as the next pass to be loaded. A discussion of this capability of having both general and specific macros in the source deck, not generated by the Parsing Algorithm, is deferred until section VII.

B. Parsing Algorithm Theory

The Parsing Algorithm converts the equation within the variable field of the LET card image into one or more general macro card images. The heirarchy of precedence of the operators, the order in which the arguments appear, and the arguments being operated on, together determine which general macros are created.

In the operation of the Parsing Algorithm, the equation is scanned from left to right on a character by character basis. It is able to decide when it has scanned enough characters to form an argument, and to recognize that the argument is bounded on either side by operators. Pushdown stacks are used to remember which arguments and which operators have been scanned. In a systematic way, the arguments and operators are tested to see if their relation to each other matches an allowed combination as specified by the Precedence Table. When appropriate combinations are recognized, general macro images are created which specify the operation and the arguments to be operated on.

The Precedence Table is shown in Appendix B along with the rules for its use which together specify the syntax for the subset of FORTRAN realized by the arithmetic compiler. All allowable relationships between argument and operator can be generated, or determined, from this table. The recursive definition approach is strictly followed in the logic of the Parsing Algorithm. Appendix A illustrates the use of the table in a simple compilation example.

C. Pushdown Stacks and Tables

Two pushdown stacks, Opt and Opn, are needed to store, respectively, the operators and arguments that have been scanned but not yet used to create a general macro. Entries and deletions occur in pairs such that an argument and its following operator are always entered and removed simultaneously from their respective stacks. Since parentheses just rearrange precedence and are not strict mathematical operators, they are just single entries which leave a corresponding void in the Opn stack.

The use of pushdown stacks greatly simplifies the logic needed to generate general macros. The latest entries into the stacks and the current argument being interrogated always make up the current general macro. There is very little programming needed to do the actual parsing. Only 128 decimal locations are needed to parse any allowable FORTRAN equation. This includes detection of errors in syntax. Less than 5% of the code required to realize the arithmetic compiler is dedicated to parsing.

Only one table is needed in the Parsing Algorithm. This is the TMPR table which keeps track of the temporary storage location mnemonics. Symbolic addresses allowed by DOPE must consist entirely of numerals or letters, with at least one letter. No special characters are allowed. For the temporary storage locations, the mnemonic $V _ _ was chosen. The _ _ are numerals that range from 00 to 99. One hundred temporary locations are possible within a program, but rarely will $V05 be exceeded even for the most complex equation. Due to the TMPR table, it is the most complex equation, and not the number of LET statements, that determines how many temporary storage locations a program will require.

When the first general macro image is created, the three arguments, ARG1, ARG2, and ARG3 are Opn (I), OPND, and $VOO respectively. The temporary storage location, $VOO, is entered in TMPR. The result of the mathematical operation performed on ARG1 and ARG2 is to be stored in $VOO. In a subsequent general macro, from the same equation, $VOO will eventually appear as an ARG1 or ARG2. When it does, its purpose of saving the partial result is completed and it can be used again as an ARG3. Below is shown a typical string of general macros in which no effort is made to reuse available $V_ _ locations.

```
ADDM ALFA B    $VOO
SUBM $VOO C    $VO1
MULM DLTA ETS  $VO2
ADDM $VO1 $VO2 $VO3
DVSM F    $VO3 $VO4
EQUM $VO4 ANSW
```

Five temporary storage locations are indicated, $VOO through $VO4. The program which has the equation that generated the above

string of macros will have to be five locations longer to accomodate them.

Using the TMPR table to remember which of the $V_ _'s can be reused, would yield the following string of macros for the same equation.

```
ADDM ALFA B      $VOO
SUBM $VOO C      $VOO
MULM DLTA ETA    $VO1
ADDM $VOO $VO1   $VOO
DVSM F    $VOO   $VOO
EQUM $VOO ANSW
```

The string now requires only two temporary storage locations, $VOO and $VO1.

New temporary storage locations are created by starting at the original entry of TMPR, which is always $VOO, and scanning until the last $V_ _ is seen. For example, $VO3 could be the last entry in TMPR. The new $V_ _ will use the next highest numerical mnemonic; in this example it is $VO4. Each time a $V_ _ is used as an ARG1 or ARG2, its mnemonic is taken out of the table. Below is shown a typical TMPR table before $VO3 is used as an ARG1 or ARG2. The end of TMPR is at K = 6.

K	TMPR
0	$VOO
1	$VO1
2	$VO2
3	$VO3
4	$VO4
5	$VO5
6	

After $VO3 is used, it is removed as shown below.

K	TMPR
0	$VOO
1	$VO1
2	$VO2
3	
4	$VO4
5	$VO5
6	

The effective end of the table is now at K = 3. When there is a need for a new temporary storage location, $VO3 will be created, not $VO6.

This technique minimizes the number of possible temporary storage locations that have to be added to the program.

There is one of the $V__'s that is reserved and not affected by the TMPR table. It is $V99, the location that is needed if a MULM or DVSM macro occurs in the program. Due to the fact that DISAC's hardware multiplier/divider cannot handle signed numbers, the values of the arguments must be made positive. This temporary storage location holds the absolute value of the multiplier/divisor which keeps the original value of the argument unchanged in core.

D. Pass 1.5. and Macro Optimization

As previously stated, pass 1.5 was created expressly for the macro optimization function because there was not enough core available in pass 1. Pass 1.5 also updates the symbol table and the card image relative location values, and inserts certain symbolic addresses and the transfer vector at the physical end of the program. The size of the program is fixed at the termination of pass 1.5.

Each card image that comes through pass 1 from the source deck has associated with it a relative location value. Gaps in successive relative location values corresponding to the length of the longest specific macro were introduced between general macros. Once a specific macro is chosen to replace the general macro, the gap must be updated. Not only is the relative location of every card image affected from that image to the end of the program, but the correction factor itself is accumulative. Every time a specific macro choice is made, the correction factor must be updated.

Recognition of the END card causes a scan to be made of the symbol table to find all the symbolic addresses that do not have their relative location fields filled in. Use of the ASSN and DIMN pseudo-ops can cause this condition, as well as each time a $V__ is created in pass 1. The symbolic addresses that are found are entered at the end of the program as successive labels thus reserving a core location for each. The transfer vector is inserted immediately after the label list; the writing of the END mnemonic is delayed until the end of the transfer vector. When it is copied onto the output buffer tape, pass 1.5 is terminated and control goes back to the System to bring in pass 2.

E. Macro Optimization Theory

Optimization has many connotations. The nuance implied here is to eliminate the needless storing and loading of partial results throughout the macro string. No attempt is made to realize the equation in any other way than the programmer wrote it. Macro optimization, not equation optimization, is the intent.

The function of Macro Optimization is to convert a string of general macro card images into a string of specific macros. There

is a general macro mnemonic for each of the seven mathematical operations. It specifies the operation, and the variable field holds the arguments involved. As mentioned before, the general macro mnemonic can be considered to be the name of the set of all possible macros that perform the operation. The number of elements in each set was determined in section V.

Figure C1 is a pictorial representation of the set concept for the six independent mathematical operations. The subset structure is shown for each. The finer details of macro group and specific element are shown in Figures C2 through C5.

The macro approach yields a four-step procedure that must be used for specific macro selection. The first step is the operation determination which has already been discussed; it is performed by the Parsing Algorithm when it selects a general macro mnemonic. Macro Optimization realizes the remaining three steps.

The second step is concerned with the positioning of the input arguments according to an evaluation criterion that has been established. This is the #l.s. parameter zero criterion discussed in section VA. All elements of a general macro set can be segregated into subsets according to how the input arguments are shifted.

The third step takes into account how the input arguments are obtained. Within each subset there are groups of elements according to whether the input arguments are obtained from core or one of the operational registers.

The final step is concerned with the depositing of the result. Within a macro group there can be three specific elements. They differ from each other only in what they do with the result of the operation. The possibilities are to leave it in the AC, the MQ, or to deposit it in core.

In summary, Macro Optimization makes a series of choices. A macro subset is selected from the general macro set based entirely on the SIFW word parameters of the arguments in the variable field. A macro group is chosen from the subset with the result of the previous macro taken into account. The final choice is the selection of a specific macro element from the group based on the input to the following macro.

F. Partition Concept of Specific Macros

The set structure implies that the specific macro code consists of parts that are independent of each other. The sole determination of macro subset for the K^{th} macro of a string is based on the SIFW word parameters of its arguments. Exactly where the arguments come from or where the result is to be stored does not affect the subset choice. The operation, the input, and the output portions

of the code are equally independent of each other. This characteristic is described as the partition concept and is illustrated below. The AD10 specific macro is used as the example.

```
AD10  LDMQ ARG1                    Partition 1
      LDAC ARG2
      ---------------------------
      QLS  PAR1
      ALS  PAR2                    Partition 2
      AACQ
      AQRS PAR3
      ---------------------------
      STMQ ARG3                    Partition 3
```

This version of AD10 is slightly different from the one shown in Appendix C. It is rearranged above to illustrate a visible division between the partitions. They have equivalent functional divisions.

Partition 1 obtains the input arguments, partition 2 performs the mathematical operation, and partition 3 stores the result. Taking AD10 as the K^{th} macro of a string, it can be seen that the code of partition 2 cannot be reduced. There is the possibility for instruction reduction for partitions 1 and 3, however, depending on the preceding and following macros. Below is shown a portion of a string before the optimization process is begun. The illustration assumes both additions are represented by AD10 for simplicity. This string could be generated from the section of the following equation.

LET ... ... B + C + D + ...

```
K^th - 2
_____

                AD10   LDMQ B
                       LDAC C              Partition 1
                       ---------------------------
                       QLS  PAR1
                       ALS  PAR2
K^th - 1               AACQ                Partition 2
                       AQRS PAR3
                       ---------------------------
_____                  STMQ $VOO           Partition 3
                       ---------------------------
                AD10   LDMQ $VOO
                       LDAC D              Partition 1
                       ---------------------------
                       QLS  PAR1
K^th                   ALS  PAR2
                       AACQ                Partition 2
                       AQRS PAR3
                       ---------------------------
_____                  STMQ $VOO           Partition 3
                       ---------------------------

K^th + 1
```

The last thing partition 3 does is to store the result of its operation in $VOO and the first thing that partition 1 of the following macro does is load that result from location $VOO. At the end

of partition 2 of the K^{th} - 1, the result is in the MQ. At the beginning of partition 2 of the K^{th}, that result is back in the MQ. Both the store and load (STMQ $VOO and LDMQ $VOO) can be eliminated without affecting the performance of either mathematical operation. Just in this one boundary alone, there can be a savings of two memory locations in length and four machine cycles in speed. This result is consistent with the definition of optimization stated in section IA.

G. Pass 2 and Macro Expansion

The primary responsibility of pass 2 is to use the symbol table generated by pass 1, and its own opcode table, to form the 24 bit machine language word for each mnemonic card image read from the input buffer tape. This function is performed in an 'appropriate evaluation routine' depending on which mnemonic is in the instruction field. If a specific macro is detected, then control is transferred to the Macro Expansion routine.

There are 54 specific add macros, 54 subtraction, 9 multiplication, 9 division, 5 equality, and 15 subroutine call specific macros. This is a total of 146 specific macro mnemonics that must appear in a comparison table to determine if the input image is in fact a specific macro. The checking procedure would be time-consuming since each input image would not only have to be compared with the opcode table to find the appropriate evaluation routine, but also with the 146 images of the macro comparison table. The opcode comparison table is unavoidable, the macro table is not.

At the time it was presented, there seemed to be no reason for choosing the _D_ _, _E_ _, and _F_ _ element distinction within a group and the use of the numerical digits in the last two positions of each mnemonic. By doing so, the 146 comparisons can be reduced to six mask and compare operations with a macro comparison table of length six. The BCD characters D, E, and F differ from each other in the six bit numerical code by just the final two bit positions; D = 010100, E = 010101, and F = 010110. All BCD numerals can be distinguished from alphabetic characters expressed in this code in a similar fashion. Combining this with the unique first character of each mnemonic, the macro comparison table need contain only six entries; AD_ _, SD_ _, MD_ _, DD_ _, CD_ _, and QD_ _. The _ _ _ notation means the bit pattern of the remaining 24 bit word of each mnemonic is zero. The mnemonic on the card image is masked so that if the second BCD character is a D, E, or F it will become a D, and if the last two BCD characters are numerals, they will become a pattern of all zeros. An alphabetic character in those positions will not yield a pattern of all zeros and no match will be found on the macro comparison table. Using this mnemonic scheme yields a worst case savings of approximately 76% in time for the scanning procedure and a savings of approximately 140 decimal words of pass 2 core which is at a premium.

The opcode table consists of two data lists. The first, LIST1, contains every allowable mnemonic that can be placed in the instruction field of a card image. When a match is found through a scanning process, the data word is obtained from the corresponding position in the second list, LIST2. Twelve bits of this 24 bit word contain the address of the appropriate evaluation routine and the remaining 12 bits contain the bit pattern for the opcode portion of the machine language equivalent to the card image instruction.

H. Macro Expansion Theory

Macro expansion is necessary in pass 2 because there is not enough core in pass 1 to hold the BCD expansion of each of the 146 specific macros. If this were possible, then instead of specific macro mnemonics, a string of card images would be generated with a single BCD assembler instruction per image. Pass 2 would then be unable to distinguish the generated assembler images from the ones that appear in the source program and would only have to determine the appropriate evaluation routine to use for each assembler instruction.

Under normal use, pass 2 expects to see BCD instruction mnemonics and BCD symbols in the address field of the card image. It can write the BCD card image directly on the output buffer tape as a part of the listing. It uses the opcode table to transform from BCD mnemonic to numerical machine code for the instruction field. This constructed machine code instruction is written on the output buffer tape as a part of the object deck. BCD instruction mnemonics require 24 bits (1 core word) of storage, six bits per character. The machine code equivalent can be contained within 12 bits. Since core is scarce in pass 2, it becomes advantageous to store the expansion of each macro in machine code. This would leave 12 bits per word for a coding scheme indicating how the address portion of each expansion instruction is to be filled in.

Storing the machine code means that the BCD equivalent of each instruction must be obtained to place in the listing. This is accomplished by using the opcode table in the reverse direction than which it was originally designed. A 12 bit match is sought in LIST2 and the corresponding BCD mnemonic is obtained in LIST1. The mnemonic is placed in the instruction field of the card image that is being built in the CBUF buffer.

The address to the appropriate evaluation routine is retained during the scan of LIST2. With the deciphering of the coding scheme in the remaining 12 bits of the expansion instruction, the address field of the card image in CBUF is filled in with BCD information. Control is then given to the evaluation routine pointed to by the LIST2 address and the card image is evaluated just as if it had originally appeared in the source deck. The evaluation routine finishes the task of forming the 24 bit machine instruction equivalent for the object deck, then writes this machine language and the BCD card

image on the output buffer tape as part of the object deck and the listing respectively.

I. Core Requirements for Specific Macros

Appendix C holds the expanded versions of all 146 specific macros. Even with the storing of the numerical machine language equivalent and the address field coding scheme, one word of core per instruction is required. It would take approximately 1200_{10} core locations to hold all the macros; this still exceeds the available locations. The problem is solved by a technique called 'selective loading'.

There is similarity in the code for all the macros of a given set and also similarities between certain pairs of sets; ADDM and SUBM, MULM and DVSM, and CALM and EQUM. It is possible to map any one of the specific macro elements into one of these three categories; CASE1 for ADDM and SUBM, CASE2 for MULM and DVSM, and CASE3 for CALM and EQUM. It is possible to associate with each CASE_ a list of machine instructions from which any one of the specific macros in that particular category can be formed. Assuming this is possible, each specific macro would have a control word associated with it to point to which instruction within the list is to be used in forming the code for the macro.

The format for the control word is that every bit within the control word is associated with one instruction within the list. The left most bit is associated with the first instruction in the list and the fifth bit from the left is associated with the fifth instruction in the list. Care must be taken in the forming of the CASE_ list since the order of instructions for all macros within the category must be preserved within the single list. The control word dictates which instructions are to be deleted or used proceding from top to bottom of the list. There is no order information in the control word; order has to be inherent in the list. This is one reason why more than a single list is needed for all 146 specific macros. A second reason is that control words have only 24 bits which implies that it has control only on the first 24 instructions of the list. Every macro within a category must be able to be formed by controlling the first 24 instructions.

A '1' in a control word bit position dictates that the corresponding instruction is to be deleted in the forming of the specific macro; a '0' means it is to be used. The actual control word is not limited to 24 bits in the general case. It can be considered to be a variable length word of which the left most 24 bits can be '1' or '0' and the rest of it all zeros. The length of the control word is equal to the length of the list. If there was a macro (fictitious example) whole list was 124 instructions long, the control word would have 124 bits. Only the first 24 could be specified; the last 100 bits would be all '0' which would cause the last 100 instructions from the list to be used in the generation of this macro.

Since a single list for all macros would have to be different in more than the first 24 positions to accommodate all 146 macros, three CASE_ lists are needed and are shown below.

CASE1

```
LDMQ ARG1
LDMQ ARG2
LDAC ARG2
QLS  PAR1
AQRS PAR1
QLS  PAR2
AQRS PAR2
LDAC ARG2
LDAC ARG1
ALS  PAR2
ALS  PAR1
AACQ
SACQ
AMQA
SMQA
ADMQ ARG2
SBMQ ARG2
ADMQ ARG1
SBMQ ARG1
PACQ
CHSQ
AQRS PAR3
NOP
```

CASE2

```
LDAC ARG2
PMQA
TACP *+2
CHSA
ALS  PAR2
STAC $V99
LDAC ARG2
PMQA
LDMQ ARG1
EMQA
ARS  23
SRAC *+6
TMQP *+2
CHSQ
ZAC
QLS  PAR1
LLS  PAR1
DPDV $V99
MPY  $V99
PACQ
PADA **
TACZ *+2
CHSQ
AQRS PAR3
NOP
```

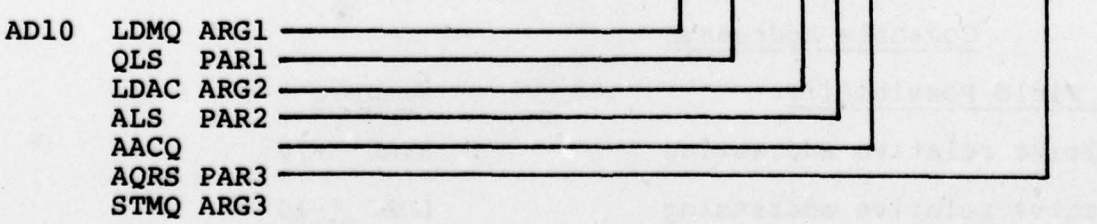
CASE3

```
LDMQ ARG1
AQRS PAR1
LDMQ ARG2
AQRS PAR2
QLS  PAR1
QLS  PAR2
CALL ARG1
AQRS PAR3
NOP
```

The NOP position in each CASE_ is filled in with STMQ ARG3, PMQA, or left blank depending on whether the specific macro is a _D_ _, _E_ _, or _F_ _ respectively.

An example is shown below illustrating the form and use of the control word in generating a specific macro from a CASE_ list.

Specific Macro	Control Word	Expanded Control Word
AD10	33527773	01101110101011111111011



The saving in core made by using the selective loading technique makes the arithmetic compiler for DISAC feasible. There needs to be two tables each 52 locations long. The first table contains group mnemonic, _D_ _, for each macro. The second is a corresponding table of the control words. A single control word can be used for all three elements since the _D_ _, _E_ _, _F_ _ information just affects the entry into NOP. The CASE lists require only 58 core locations and the two tables require (2 * 52) just 104 core locations. Therefore approximately 162 locations are used in the storage of the expansion of all 146 specific macros. The remaining core of the available 920 locations can be used for the code that builds up the macros and interprets the address coding scheme.

J. Address Field Coding

A macro is made up of the same assembler instructions that the programmer has available to him when he generates his source deck. The coding scheme must be capable of representing any allowable address field in the remaining 14 bits of the 24 bit word. Earlier, the breakdown between the machine code equivalent instruction field and the address field was 12 to 12, but only 10 bits are needed for the opcode. Reference 1 section 4 gives a detailed discussion of the format within each machine instruction. When the search is made for the BCD equivalent in the opcode table, two zero bits are joined to the 10 bit opcode to make it a 12 bit search. These two bits are reserved in the coding scheme to specify which index register has been used in the instruction.

The object is to represent BCD address fields that could be a maximum of six BCD digits long (36 bits) in the available 14 bits of the expansion instruction word. If saving core was not the object, then two core words per instruction could be used. Twelve of the 14 available bits of the first word could hold two BCD digits and the 24 bits of the second word could hold the remaining four. Using 50 words of pass 2 core to represent a 25 word macro is not acceptable.

The coding scheme can make efficient use of the 14 bits to represent nearly all possible address fields. Those that can not be coded require the two-word format. Below is shown two tables of possible address field requirements and an example of each. The first table is codeable, the second is not. With the exception of subroutine call, and the $V99 symbolic address, the other two entries of the second table never appear in a mathematical macro.

Codeable Addresses

Address Field Possibility	Example
1. Positive relative addressing	STAC *+79
2a. Negative relative addressing	LDAC *-15
b. Relative addressing indexed	STMQ *+13,3

3a.	Unspecified address	LDMQ **
b.	Unspecified address indexed	LDXR **, 1
4.	No address	PMQA
5.	Reference to macro argument	LDMQ ARG1
6.	Reference to macro parameter	QLS PAR2

Non Codeable Addresses

Address Field Possibility		Example
7a.	Relative addressing greater than 79	STAC *+89
b.	Call to subroutine	CALL SQRT
c.	Symbolic address	STMQ TEMP
d.	Symbolic address indexed	STMQ TEMP,2

There are seven address possibilities shown which can be expressed in terms of what information is placed in the address field of the card image being built in CBUF.

1. Entry of the two BCD symbols, *+, and the remaining 9 bits of the word interpreted as two BCD characters.

2. Entry of the two BCD symbols, *-, and the remaining 9 bits of the word interpreted as two BCD characters.

3. Entry of the two BCD symbols, **, that are in the remaining 12 bits of the word.

4. Entry of a blank address field.

5. Entry of the BCD form of the macro argument pointed to by the remaining 9 bits of the word interpreted as a binary number.

6. Entry of the BCD form of the macro parameter pointed to by the remaining 9 bits of the word interpreted as a binary number.

7. Entry of six BCD characters, two of which are in the remaining 9 bits of the word and four of which are in the next consecutive memory location.

Three bits are needed to represent which of the seven states is indicated, which leaves only nine bits of information. The maximum number that can be expressed in 9 bits is 79 which explains the difference between line (1) of the codeable addresses and line (7a) of the non codeable addresses. The two index bits that have been ignored so far are used to specify one of the three index registers. If the bits are nonzero, then a comma and the BCD number equivalent to the two bits are entered in the card image address field.

The specification for a macro argument or parameter is done by a binary number from 1 to 3 for an argument and a parameter. The number refers to ARG1, ARG2, or ARG3 respectively and PAR1, PAR2, or PAR3 respectively as they appeared on the input specific macro card. This image held the BCD forms of all arguments and parameters. The ARGs' position and the PARs' BCD values are placed in separate tables from which their BCD forms can be transferred to the assembler instruction card image being built in BLOC. A negative PAR3 for an ARG3 causes a right shift to become a left shift, and a zero PAR3 causes the elimination of the instruction from the expansion.

With the writing of the last instruction of the last LET statement onto the output buffer tape, the arithmetic compiler function of DOPE is complete. Control returns to the System when the END card is recognized; the output listing and object deck are stored on the output buffer tape.

CHAPTER VII

RESULTS AND CONCLUSIONS

A useful fixed point arithmetic compiler can be implemented on a mini computer with the macros approach realization. The objectives and criteria established in section I have been met. A brief summary of the merits and deficiencies of the general design and the DISAC implementation is offered below.

A. Areas of Improvement

There are five major areas where the compiler could be improved; three for the arithmetic compiler in general and two for the specific DISAC implementation. It would be convenient to be able to have constants (numbers) appear directly in the LET statement. Numbers with no binary point specified would be assigned the BMT parameters calculated from the number of decimal digits to the left and right of the decimal point; the binary point would be calculated to make the #l.s. parameter equal to zero. Binary point specification would override the #l.s. parameter zero criterion. Numbers that are multiplicative factors would be checked to see if they are powers of 2. If so, the product would be reduced to a left or right shift. Numbers that are exponents would be checked to see if they were small integers. If so, successive multiplications or divisions would be implemented instead of the subroutine calls.

A second area of improvement would be to use a more rigorous rule for the calculation of the T fields for the intermediate results of the macros and the results of the library subroutines.

A third area to be improved in the general design would be to realize optimization across macros. This can be done while still avoiding equation optimization. The situations can arise where an intermediate result could be left in the AC of the K^{th}-1 macro, the K^{th} macro could perform a manipulation between the MQ and core, and the K^{th}+1 macro combine the results in the AC and MQ. These situations are rare, but they would be useful to realize.

A subtle feature occurs in the DISAC implementation that wastes symbol table space. Three words of core are required for every symbolic address that appears as a LET statement argument. This is necessary, but the three-word set is also reserved for every other symbolic address in the program that appears as a label. This is a loss of approximately 30% of the total symbol capability that a program can have. With the three-word format, programs are restricted to approximately 150 symbols. If a two-word format could be

implemented for non LET statement symbolic addresses, this would increase to over 200 symbols.

The second shortcoming for DISAC is the three pass compilation process. A two pass compiler would not only be faster with less wear on the tape units, but would be less susceptible to input and output errors.

B. Useful Features

Although the arithmetic compiler has a primary function to convert LET statements into machine code, the macro approach of the Parsing Algorithm, Macro Optimization, and Macro Expansion has additional flexibility inherent in its structure. The Macro Optimization routine just looks for and operates on general macro images. It assumes they come from the Parsing Algorithm, but it cannot distinguish one so generated from one entered directly in the source deck. This capability in conjunction with the CORR pseudo-op, gives the programmer direct control over the general macro level of the DOPE generated Code. Inserting general macros still requires pass 1.5 to generate the shift parameters and choose a specific macro.

Macro Expansion of pass 2 expects to see specific macros generated by Macro Optimization, but it also does not care where they come from. The programmer can also enter specific macro cards in his source deck. In doing so, he must calculate his own shift parameters, but he can avoid pass 1.5.

These two natural extensions to the macro approach make the arithmetic compiler more powerful than the original design specified.

APPENDIX A

Compilation Example

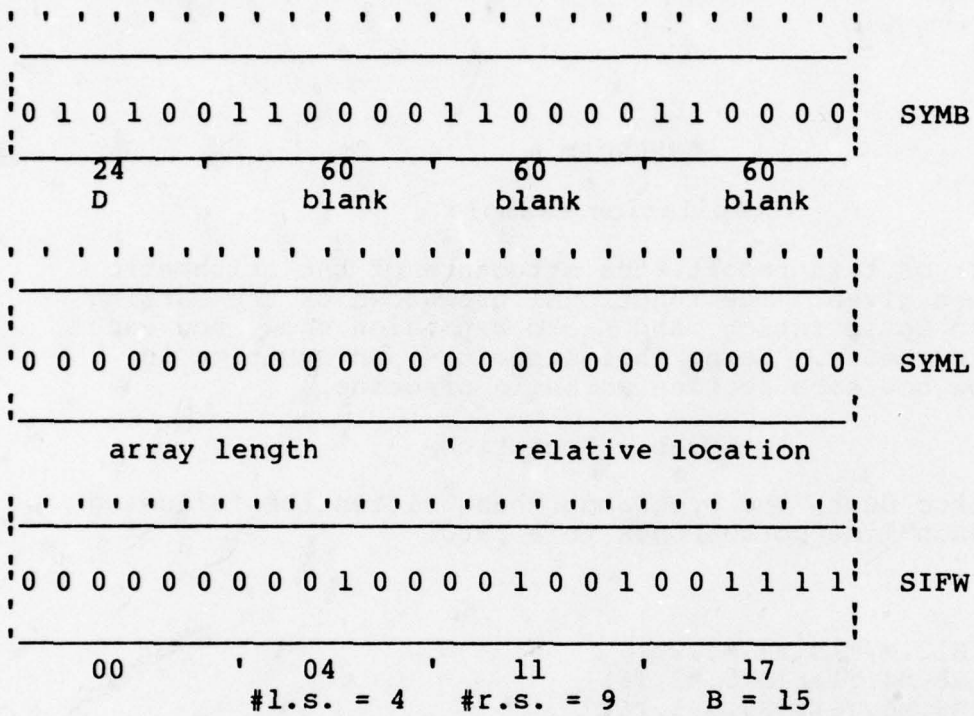
In the text of this report, the structure of the arithmetic compiler has been given. The functional breakdown of the Parsing Algorithm, Macro Optimization, and Macro Expansion shows how each routine works in theory. Using this structure, an equation is compiled to show how each routine works in practice.

A. Pass 1 Operation

In the source deck, the programmer has written the following program where each line corresponds to a card.

```
        JOB
        DOPE
        ASSN A(B12,M,T)B(B6,M2,T3)
        ASSN C(B8,M4,T1)D(B15,M3,T6)
        ASSN E(B6,M6,T6)F(B9,M-3,T6)
STRT    LDMQ A
        SRMQ LOAF
CLYD    LET A=(((B+C)*D)+E)*F
LOAF    PADQ **
        SRMQ *-1
        HALT
B       DEC 3.125B6
C       DEC 15.5B8
D       DEC -5.172B15
E       DEC 53.8B6
F       DEC .1B9
        END
```

Pass 1 operates on this program to produce a symbol table. The ASSN statements are evaluated first and store the BCD form of the six arguments in their respective SYMB locations. The SYML word is left untouched as the BMT data is entered into the SIFW word of each. Two equations from section IIIC1 are reproduced here to show the conversion from MT parameters of the ASSN statement to the #l.s. and #r.s. parameters stored in the SIFW word. The three-word set that was introduced in section IIIC is shown below for the argument D. The other five arguments have similar entries in the symbol table.



#l.s. = 22 - (B+M) = 22 - (15+3) = 4

#r.s. = B - T = 15 - 6 = 9

B = B = 15

The array length and relative location fields of SYML are filled in when the argument is recognized as a label. Since none of the arguments in the ASSN statements are dimensioned variables, array length will be set to 0001.

The symbol STRT is put in the symbol table and given the location 0000; CLYD is given location 0002. At this point in pass 1, the Parsing Algorithm is entered and the evaluation of the LET statement is begun. When it terminates, there is a gap in the relative location values between the symbol CLYD and the symbol LOAF. Space is reserved for the object Code to be inserted by Macro Expansion of pass 2. In this specific case, 67_8 locations are reserved. The symbol LOAF is given a relative location of 0071_8. Beginning with B at 0074_8, the consecutive relative locations are entered into the SYML words of the arguments already created by the ASSN pseudo-ops.

B. Interpretation of the Equation

Scanning for arguments in the equation is done from left to right. The variable field of the LET statement is stored in a buffer

called CBUF. Each individual character of the equation can be referenced by indexing CBUF with J. If J is set to 15, then CBUF(J) would point to the 15^{th} character in the equation. A similar index I is used to reference the entries of the Opn and Opt stacks. This indexing is used in the flow diagrams, the following discussions, and in the Precedence Table of Appendix B.

1. A scanning of CBUF for an operator terminates with CBUF(J) pointing at an =, J = 3. All characters that came before J = 3 and after the last seen operator constitute a variable. Since the previous operator was the blank equation delimiter, the object A is defined to be a variable. Variable appears twice on the right side of the table in Appendix B. The default match appears in line (8) and a following operator match in line (12). Adhering to the precedence of a match as stated in rule b, the following operator match is realized. Rule f causes Opn and Opt to appear as shown.

I	Opn	Opt
I	A	=

2. Continuing at J = 4, the scan terminates at J = 8 with the three open parentheses being stored in Opt and the object B being defined as a variable. The following operator can be referenced as CBUF(J) just as the preceding operator in Opt(I); I and J having their current values.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
I	A	=	(	B	+
2		(			
3		(			
4		(			

3. With B as the object, CBUF(J) is the + and Opt(I) is the (. A match for these combinations is sought on the table. The object is first a variable. The default option has to be taken, which redefines the object as a primary. The primary appears three times on the right side of the table. CBUF(J) is not a ** so there is no following operator match, and Opt(I) is not a **. The only match is the default which redefines the object to be a factor. Factor appears twice on the table. Since the object is not preceded by a * or /, it is redefined to be a term by default. Term appears twice on the table. The object again only matches the default condition for a term since it is not preceded by a + or -. The object is now an arithmetic expression which appears three times on the table. Beginning at the top and checking for following operator matches first, it is seen that the object is not followed by a). CBUF(J) does equal a + so the following operator match is finally found. This object and the following operator are stored respectively on the Opn and Opt stacks.

I	Opn	Opt
1	A	=
2		(
3		(
4		(
5	B	+

4. Continuing at J = 9 and terminating at J = 10, a new object is scanned for. The object is C and has CBUF(J) equal to).

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	+	C	)
2		(			
3		(			
4		(			
5	B	+			

5. The object is now C, its following operator is a), and its preceding operator is the +. This object defaults down to a term where a match in the table is found with the preceding operator. Rule d states that a preceding operator match causes the latest entries to be erased from the stacks and a macro to be outputted. The macro is a general macro card image incorporating the present object, Opn(I), and the current temporary storage location.

Output Macro					I	Opn	Opt
CLYD	ADDM B	C	$VOO	$	1	A	=
					2		(
					3		(
					4		(

6. $VOO is the new object that the table operates on and is defined as an arithmetic expression corresponding to the defining line where the match was found. With the removal of the latest entries from the stacks, Opt(I) becomes (. Note J is still equal to 10.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	(	$VOO	)
2		(			
3		(			
4		(			

7. With $VOO as the object, a parenthesis match is found which is treated as a special case. Since no mathematical operation is performed by a set of parentheses, no macro is outputted. I is decremented thereby removing the latest entry on Opt which was the preceding operator, and J is incremented to skip over the following operator. With the set of parentheses removed, the object is still $VOO but is now a primary via the line the parenthesis match was found on.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	(	$VOO	*
2		(			
3		(			

8. The object $VOO defaults down to a term and the following operator match is found. CBUF(J) and $VOO are entered in the stacks.

I	Opn	Opt
1	A	=
2		(
3		(
4	$VOO	*

9. The scan for the next object is begun with J = 12 and terminates with J = 13. The new object D is considered first to be a variable.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	*	D	)
2		(			
3		(			
4	$VOO	*			

10. D defaults down to a factor and the preceding operator match is found. The latest entries on the stacks are removed, a macro is outputted, and the temporary storage location mnemonic becomes the new object. Since an optimizing routine in the Parsing Algorithm minimizes the number of temporary locations created, $VOO can be reused in the general macro as the temporary location mnemonic.

Output Macro		I	Opn	Opt
MULM $VOO D	$VOO $	1	A	=
		2		(
		3		(

11. $VOO is the new object. It is a term which defaults down to an arithmetic expression. With the removal of the latest entries on the stacks, Opt(I) is now the (. The index J is still equal to 13; CBUF(J) is the).

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	(	$VOO	)
2		(			
3		(			

12. $VOO is the object and has the (as its preceding operator with the) as its following operator. This is the same circumstance as in step (7) above. The result is to increment J to look beyond the closed parenthesis, and to remove the latest entries from the stacks. With J now equal to 14, CBUF(J) is the +. Opt(I) is now (. $VOO is the new object now defined as a primary.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	(	$VOO	+
2		(			

13. $VOO defaults down to an arithmetic expression and the following operator match is found. The entries are made into the stacks.

I	Opn	Opt
1	A	=
2		(
3	$VOO	+

19. The scan for the next object is initiated with J = 18 and terminates with J = 19 at the end of the equation. F is the new object which is subjected to the table as a variable.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	*	F	
2	$VOO	*			

20. F defaults down to a factor where the preceding operator match is found. This is the same situation as in step (10) above.

Output Macro	I	Opn	Opt
MULM $VOO F $VOO $	1	A	=

21. $VOO is the new object. It is a term which defaults down to an arithmetic expression. J is at the end of the equation so it is not incremented. With the removal of the latest entries from the stacks, Opt(I) is now the =.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	=	$VOO	

22. $VOO is a term and it defaults down to an arithmetic expression where the preceding operator match is found. The latest entries from the stacks are removed which depletes them. This empty condition is the indicator for terminating the Parsing Algorithm. The final macro is outputted and the remainder of the program is operated on by pass 1. The END card terminates pass 1.

Output Macro	I	Opn	Opt
EQUM $VOO F $			

C. Pass 1.5 Operation

The input program for pass 1.5 is shown below for comparison with the program that was the input to pass 1.

```
                    JOB
                    DOPE
                    ASSN A(B12,M,T)B(B6,M2,T3)
                    ASSN C(B8,M4,T1)D(B15,M3,T6)
                    ASSN E(B6,M6,T6)F(B9,M-3,T6)
0000 00000000 STRT  LDMQ A
0001 00000000       SRMQ LOAF
                    LET  A=(((B+C)*D)+E)*F
0002 00000000 CLYD  ADDM B    C     $VOO  $
0010 00000000       MULM $VOO D     $VOO  $
0033 00000000       ADDM $VOO E     $VOO  $
```

```
0042 00000000          MULM $VOO F     $VOO  $
0065 00000000          EQUM $VOO A       $
0070 00000000          PADQ **
0071 00000000          SRMQ *-1
0072 00000000          HALT
0073 00000310 B        DEC  3.125B6
0074 00007600 C        DEC  15.5B8
0075 77264700 D        DEC  -5.172B15
0076 00006563 E        DEC   53.8B6
0077 00000060 F        DEC  .1B9
                       END
```

Pass 1 has made a number of additions to the original source program. The first is the insertion of the relative location values and the eight digit octal fields for each instruction. Pass 2 will insert the numeric code. Second is the addition of the general macros and the LET statement appearing as a remark. Third, the numerical values for the data have already been evaluated and entered. Fourth, the gaps left for each macro correspond to the length of the longest specific macro that could be selected by Macro Optimization; 7_8 for (+), 10_8 for (-), 3_8 for (=), 23_8 for (* and /), and 5_8 for a subroutine call.

Pass 1.5 begins operation by scanning the individual card images one at a time and writing them on the output buffer tape while searching for the general macros. If the correction factor for macro gap reduction is non zero, the symbol table relative location value and the card image relative location value are updated. If the factor is zero, no change is made.

1. The first general macro seen has the label that originally appeared on the LET statement.

```
CLYD     ADDM B     C     $VOO  $
```

Since ADDM is the first macro seen in this string there is no optimization made on its partition 1. The SIFW words are obtained from the symbol table for ARG1 and ARG2, B and C respectively. The parameters are tabulated below.

ARG	#l.s.	#r.s.	Binary Point	M	T
B	14	3	6	2	3
C	10	7	8	4	1

The adjusted binary points are compared to determine which of the specific macro subsets within the general macro set will be chosen. The addition will be done as B18 requiring C to be left shifted its maximum amount of 10. The argument B is also left shifted, but just enough to align the binary points; it is shifted 12. This is a realization of the left shift parameter zero criterion discussed in section V. The condition of left shifting both arguments

14. The scan for the next object is begun at J = 15 and terminates at J = 16 with the recognition of the operator). The new object, a variable, is E.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	+	E	)
2		(			
3	$VOO	+			

15. E defaults down to a term and the preceding operator match is found. This is similar to step (10) above.

Output Macro	I	Opn	Opt
ADDM $VOO E $VOO $	1	A	=
	2		(

16. $VOO is the new object and defaults down to an arithmetic expression. J remains the same but the removal of the latest entries causes Opt(I) to become the (.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	(	$VOO	)
2		(			

17. $VOO is the object and is in the same situation as in step (7) above. J is incremented beyond the closed parenthesis and the decrementing of I allows Opt(I) to point to =.

I	Opn	Opt	Opt(I)	Object	CBUF(J)
1	A	=	=	$VOO	*

18. $VOO is the new object, a primary, and has * as its following operator and the = as its preceding operator. $VOO defaults down to a term and the following operator match is realized. The entries are made on the stacks.

I	Opn	Opt
1	A	=
2	$VOO	*

is realized by the AX10 group within the subset of AX1X. The three shift parameters, PAR1, PAR2, and PAR3 are already determined at this stage; they are 12, 10, and 1 respectively. The result of the addition is assigned to ARG3 which is $VOO. In keeping with the criterion, the result of the B18 addition is right shifted one position, PAR3 = 1, to make its #l.s. parameter equal to zero. The set of parameters associated with $VOO are shown in step 2 below. The #r.s. parameter was arrived at by using the smallest T field of ARG1 and ARG2 which is T1 of C. The determination of which specific macro element will be chosen from the AX10 group will depend on the joint optimization of partition 1 of the next macro, $2, and partition 3 of $1. The first macro has the following form at this stage.

```
CLYD    AX10 B     C     $VOO  $1  12 10 1
```

2. The next macro in the string is the multiply macro.

```
MULM $VOO D     $VOO  $
```

The SIFW word parameters of $VOO are available and the parameters of D are obtained from the symbol table.

ARG	#l.s.	#r.s.	Binary Point	M	T
$VOO	0	16	17	5	1
D	4	9	15	3	6

Using the equation of section V for the multiplication operation, the binary point of the result is calculated to be B14. PAR1 and PAR2 are found to be 1 and 5 respectively. The result of the multiplication is assigned to ARG3 which is $VOO. The shifts of PAR1 and PAR2 assure that the result has its #l.s. parameter equal to zero. The #r.s. parameter is arrived at by using the smallest T field of ARG1 and ARG2 which is T1 of $VOO. The set of parameters associated with $VOO (as ARG3) are shown in step 3 below. The specific macro subset is already chosen to be MX1X since it is the only one within MULM. The joint optimization of partition 3 of $1 and partition 1 of $2 will determine the macro group for $2 and the specific macro element for $1. Leaving the result of the addition in the MQ instead of storing it in $VOO will yield optimum code for both macros. This decision chooses the MX11 group for $2 while picking AF10 for $1. The specific element mnemonic replaces the group mnemonic on the card image as it is written on the output buffer tape. The image is shown below. The difference update to the correction factor for the choice of AF10 from ADDM is 1 yielding a correction factor of 1.

```
CLYD    AF10 B     C     $MQ$  $1  12  10 1
```

Note that the $VOO temporary storage location has been replaced by the mnemonic MQ. This is done to indicate to the programmer

that the result is left in the MQ. The following macro, $2, expects to see the value in the MQ as its ARG1. With the addition of the shift parameters and the replacing of the macro subset with the group, the $2 macro has the following form.

```
MX11 $MQ$ D     $VOO  $2  1 5 0
```

3. The third macro in the string is the addition macro.

```
ADDM $VOO E     $VOO  $
```

The SIFW word parameters of $VOO are available and the parameters of E are obtained from the symbol table.

ARG	#l.s.	#r.s.	Binary Point	M	T
$VOO	0	13	14	8	1
E	10	0	6	6	6

The adjusted binary points of the two arguments are compared; B6 for E and B14 for $VOO. The decision is made to leave $VOO alone and to do a shift of eight positions on E to the left to make it also B14. This requires a shift parameter set of PAR1 = 0, PAR2 = 8, and PAR3 = 1. The binary point of the result will be left B13 with its #l.s. parameter equal to zero. The #r.s. parameter is chosen as the smallest of the two arguments, T1 of $VOO. The specific macro subset to accomplish this is AX8X. The set of parameters associated with $VOO as ARG3 are shown in step 4 below. The optimization procedure of partition 3 of $2 and partition 1 of $3 chooses the AX81 group for $3 and the specific element MF11 for $2. This choice leaves the result of macro $2 in the MQ which again causes the MQ substitution to appear in the image. The second specific macro written on the output buffer tape is shown below. The difference update for MF11 of MULM is 2, yielding a correction factor of 3.

```
MF11 $MQ$ D     $MQ$  $2  1 5 0
```

Note that this macro received one of its arguments in the MQ and left the result in the MQ for the following macro. This is a saving of three storage locations; the instruction to load the argument into the AC or MQ, the instruction to store the result in core, and the temporary storage location that was not needed. Below is shown the updated card image of macro $3 due to the optimization.

```
AX81 $MQ$ E     $VOO  $3  0 8 1
```

4. The fourth macro in the string is another multiplication macro.

```
MULM $VOO F     $VOO  $
```

The SIFW word parameters of $VOO are available and the parameters of F are obtained from the symbol table.

ARG	#l.s.	#r.s.	Binary Point	M	T
$VOO	0	12	13	9	1
F	16	3	9	-3	6

The binary point of the product is calculated to be B16. PAR1 and PAR2 are found to be 1 and 17 respectively. The result of the multiplication is assigned to $VOO (as ARG3) and the parameters are shown in step 5 below. The #l.s. parameter is assured to be zero by the shifts of PAR1 and PAR2 prior to the operation. The #r.s. parameter is obtained by using the smallest of the T fields of ARG1 and ARG2 which is T1. The macro subset MX1X is chosen from MULM since it is the only one within the set, and the optimization procedure is begun on partition 3 of macro $3 and partition 1 of macro $4. The result is the choice of the MX11 macro group for $4 and the specific element AF81 for $3. This choice leaves the result of the addition operation in the MQ for the multiplication to follow. The MQ mnemonic is entered on the $3 and $4 images to indicate that the result is not stored in a temporary storage location. The third specific macro to be written on the output buffer tape is shown below. The difference update for the choice of AF81 of ADDM is 2, yielding a correction factor of 5.

```
AF81 $MQ$ E     $MQ$  $3  0 8 1
```

The $4 macro image has been updated and now has the following form.

```
MX11 $MQ$ F     $VOO  $  1  17 0
```

5. The next macro in the string is the terminating equality macro.

```
EQUM $VOO A     $
```

The SIFW word parameters are available for $VOO and the parameters for A are obtained from the symbol table.

ARG	#l.s.	#r.s.	Binary Point	M	T
$VOO	0	15	16	6	1
A	--	--	12	--	--

The dashes (--) in the above table indicate that the ASSN pseudo-op did not specify M and T fields for the result that is to be stored in A. The binary point is specified, so result must be shifted to become B12 but will retain the #l.s. and #r.s. parameters that have been accumulated. The optimization of partition 3 of

$4 and partition 1 of $5 reveals a choice of QD21 for $5 and MF11 for $4. Since EQUM is the last macro in the string, the procedure of choosing a macro subset and then a macro group is eliminated. The macro preceding the equality is changed to a _D_ _ to combine its operation with the function of the EQUM. The EQUM specific macro is not written out on the buffer tape. The $4 macro is shown below as it terminates the string. The difference updates to the correction factor for MF11 and QD21 are 2 and 1, yielding a correction factor of 10_8.

```
MD11 $MQ$ F    A    $4  1 17 4
```

The optimum choice stores the result directly in A after combining the shift of the multiplication result four places to the right to align the binary point to meet the B12 requirement for A. The result of the equation is a number that will be stored in the location referred to by the symbol A with its binary point at B12, an M field parameter of 6 and a T field parameter of 1.

The optimization of the macros by Macro Optimization saved nine locations of core memory from the fact that no temporary storage locations ($V_ _'s) were needed and four store - load pairs were eliminated.

Macro Optimization resulted in four specific macro card images being written on the output buffer tape.

```
CLYD    AF10 B    C    $MQ$ $1 12 10 1

        MF11 $MQ$ D    $MQ$ $2 1 5 0

        AF81 $MQ$ E    $MQ$ $3 0 8 1

        MD11 $MQ$ F    A    $4 1 17 4
```

D. Pass 2 Operation

Pass 1.5 has generated an output buffer tape which is the input to pass 2. This program is shown below for comparison with the original source deck and the input program to pass 1.5.

```
                    JOB
                    DOPE
                    ASSN A(B12,M,T)B(B6,M2,T3)
                    ASSN C(B8,M4,T1)D(B15,M3,T6)
                    ASSN E(B6,M6,T6)F(B9,M-3,T6)
0000 00000000 STRT  LDMQ A
0001 00000000       SRMQ LOAF
                    LET  A=(((B+C)*D)+E)*F
0002 00000000 CLYD  AF10 B   C    $MQ$  $1  12 10 1
0010 00000000       MF11 $MQ$ D   $MQ$  $2  1 5 0
0031 00000000       AF81 $MQ$ E   $MQS  $3  0 8 1
0035 00000000       MD11 $MQ$ F   A     $4  1 17 4
```

```
0060 00000000 LOAF    PADQ **
0061 00000000         SRMQ *-1
0062 00000000         HALT
0063 00000310 B       DEC  3.125B6
0064 00007600 C       DEC  15.5B8
0065 77264700 D       DEC  -5.172B15
0066 00006563 E       DEC  53.8B6
0067 00000060 F       DEC  .1B9
0070 00000000 A
0071 00000000 $V99
                      END
```

The differences that exist between the input program of pass 1.5 and this program can be seen in the above listings. The first difference is the specific macro mnemonics have replaced the general macros. Second, the shift parameters have been added. Third, the gaps between successive macros have been updated to reflect the optimization performed by pass 1.5. Fourth, two additional core locations have been reserved at the physical end of the program. The first location is for the symbol, A, that appeared in the ASSN pseudo-op. The second location is for $V99 that is required by the multiplication macro. It is emphasized that the gaps between the macros correspond to the exact length of each macro and no longer to the largest macro of the set.

The control words for each of the specific macros are obtained and the designated instructions are obtained from CASE_. After the address fields are filled in via the coding scheme, source instructions and the CASE_ expansion instructions are indistinguishable. Control is given to the appropriate evaluation routine. It uses the symbol table for memory reference instructions, and uses BCD to binary conversion routines for numerals appearing in address fields. All information of the result of the assembly process is contained in the output listing shown below.

```
                      JOB
                      DOPE
                      ASSN A(B12,M,T)B(B6,M2,T3)
                      ASSN C(B8,M4,T1)D(B15,M3,T6)
                      ASSN E(B6,M6,T6)F(B9,M-3,T6)
0000 20400070 STRT    LDMQ A
0001 63200060         SRMQ LOAF
                      LET  A=(((B+C)*D)+E)*F
                      AF10 B    C     $MQ$  $1  12 10 1
0002 20400063 CLYD    LDMQ B
0003 51000014         QLS  12
0004 20200064         LDAC C
0005 56000012         ALS  10
0006 12500000         AACQ
0007 54000001         AQRS 1
                      MF11 $MQ$ D   $MQ$  $2 1 5 0
0010 20200065         LDAC D
0011 04100013         TACP *+2
```

```
0012 12240000          CHSA
0013 56000005          ALS  5
0014 62600071          STAC $V99
0015 20200065          LDAC D
0016 13100000          EMQA
0017 57000027          ARS  23
0020 62600026          SRAC *+6
0021 06100023          TMQP *+2
0022 13440000          CHSQ
0023 51000001          QLS  1
0024 73400071          MPY  $V99
0025 12400000          PACQ
0026 15200000          PADA **
0027 05000031          TACZ *+2
0030 13440000          CHSQ
                       AF81 $MQ$ E    $MQ$   $3 0 8 1
0031 20200066          LDAC E
0032 56000010          ALS  8
0033 12500000          AACQ
0034 54000001          AQRS 1
                       MD11 $MQ$ F    A      $4 1 17 4
0035 20200067          LDAC F
0036 04100040          TACP *+2
0037 12240000          CHSA
0040 56000021          ALS  17
0041 62600071          STAC $V99
0042 20200067          LDAC F
0043 13100000          EMQA
0044 57000027          ARS  23
0045 62600053          SRAC *+6
0046 06100050          TMQP *+2
0047 13440000          CHSQ
0050 51000001          QLS  1
0051 73400071          MPY  $V99
0052 12400000          PACQ
0053 15200000          PADA **
0054 05000056          TACZ *+2
0055 13440000          CHSQ
0056 54000004          AQRS 4
0057 63600070          STMQ A
0060 15400000 LOAF     PADQ **
0061 63200060          SRMQ *-1
0062 00000000          HALT
0063 00000310 B        DEC  3.125B6
0064 00007600 C        DEC  15.5B8
0065 77264700 D        DEC  -5.172B15
0066 00006563 E        DEC  53.8B6
0067 00000060 F        DEC  .1B9
0070 00000000 A
0071 00000000 $V99

                     END
```

APPENDIX B

Parsing Algorithm Precedence Table

The Precedence Table describes the syntax of FORTRAN equations that is realized by the arithmetic compiler. The descriptions are in the form of recursive definitions. The Parsing Algorithm of pass 1 is an implementation of the table and the rules that govern its use. Each character or string of characters in the equation must fit one of the definitions or a syntax error will result. This table detects and rejects incorrectly written equations as it determines which mathematical operations are to be performed on which arguments and in what order they are to occur.

Two pushdown stacks, Opn and Opt respectively, are used to remember objects and operators that have been scanned, but not yet used in a general macro image. Object is used to refer to a particular argument as it is continually redefined in the search for a preceding or following operator match.

Line (7) of the table, the number definition, is not realized in this version of the arithmetic compiler. No constants can appear in an equation. All mathematical operations must be between symbolic addresses. Examples of equations are shown below followed by the table and its governing rules.

```
LET A=2* (BETA+ZETA)/3                          not allowed

LET A=TWO*(BETA+ZETA)/THREE                         allowed
```

A. Precedence Table

1.	letter	A¦B¦C ··· ··· X¦Y¦Z¦$
2.	digit	0¦1¦2¦3¦4¦5¦6¦7¦8¦9¦
3.	character	letter¦digit¦(no character)
4.	operator	+¦-¦*¦/¦**¦=¦sub( )
5.	special operator	(¦)¦,¦.¦(blank)
6.	variable	character letter character character
7.	number	digit¦digit digit ··· digit

8.	primary	variable ¦ (arithmetic expression) ¦ sub(arithmetic expression)
9.	factor	primary ¦ primary**primary
10.	term	factor ¦ term {* /} factor
11.	arithmetic expression	term ¦ arithmetic expression {+ -} term
12.	arithmetic assignment statement	variable=arithmetic expression

B. Governing Rules

Rule a. An object is a string of alphanumeric characters as defined by line (3) that is separated from other arguments by operators as defined in lines (4) and (5). Each object has both a preceding and a following operator as they appear in the LET statement equation.

Rule b. Once an object is established as a variable via line (6), it is subjected to the right side of the table beginning at line (8). Proceding from line (8) to (12), the following and preceding operators are compared to the allowable operators that the object can have as it is currently defined. Every line except (12) has a default match which allows the object to be progressively redefined from variable to primary, primary to factor, etc., until a non default match is found.

Rule c. A following operator match causes the object to be placed on the Opn stack and the operator on the Opt stack. The next argument in the equation is scanned for while storing all intermediate operators on Opt. This argument is the new object and is subjected to the table via rule a.

Rule d. A preceding operator match causes the creation of a general macro using the preceding operator, the present object, the latest entry on Opn, and a $V_ _ from TMPR. The preceding operator and the Opn entry are removed from their respective stacks. The $V_ _ is the new object and is subjected to the table via rule a.

Rule e. A syntax error results if no match is found in the table for the object.

Rule f. The process terminates with the preceding operator match of line (12).

APPENDIX C

Specific Macro Expansions

Figure C1 shows the set structure for each of the independent mathematical macros. Figures C2 through C5 illustrate in detail the association of the specific elements to group, groups to subset, and subsets to set.

For all of the general macro sets, the solid lines in the figures are the boundaries to the subsets. The _X_X notation identifies the subsets where the first _ is either an A, S, M, D, C, or Q and the second _ is a number from 1 to 8. They differ from each other in the manner in which the input arguments are shifted. Section V explains the decisions that were made to determine the number of specific elements for each set.

The groups within a subset are distinguished from each other by the last character of the four-character mnemonic. It can be 0, 1, or 2. The '0' character indicates that all elements in the group expect to see input arguments in core. The '1' character indicates that ARG1 is in an operational register with the ARG2 in core. The '2' character indicates that ARG2 is in one of the operational registers and the ARG1 is in core.

The elements within a group are identical except for what they do with the result of the operation. The second character of the mnemonic is used to distinguish them. The _D_ _ will store the result in a core location referred to by ARG3, the _E_ _ will leave the result in the AC, and the _F_ _ will leave the result in the MQ.

The expanded code for each specific macro of each set is shown immediately following the figures below.

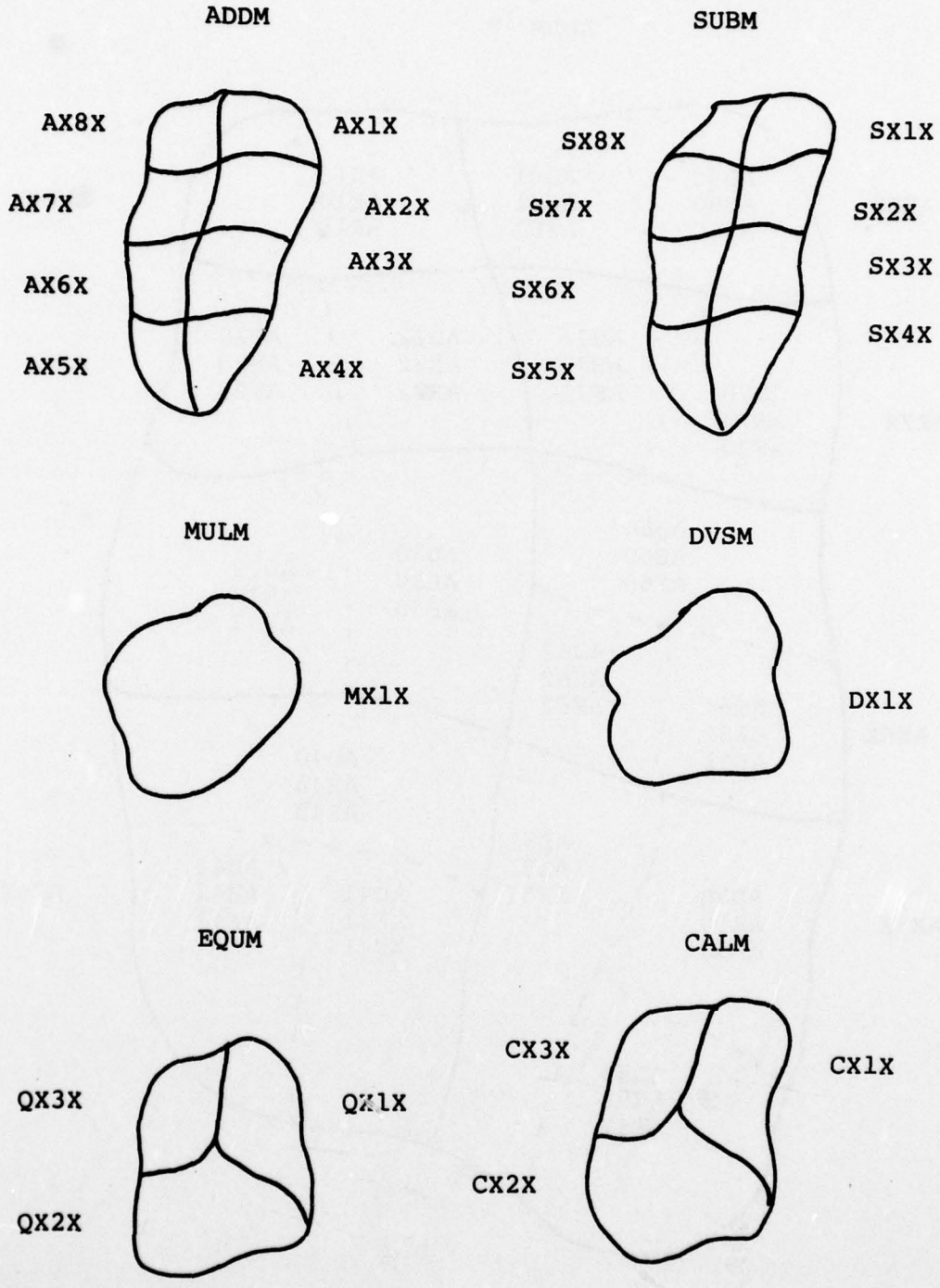


Figure C1 ARITHMETIC COMPILER SETS WITH SUBSETS

ADDM

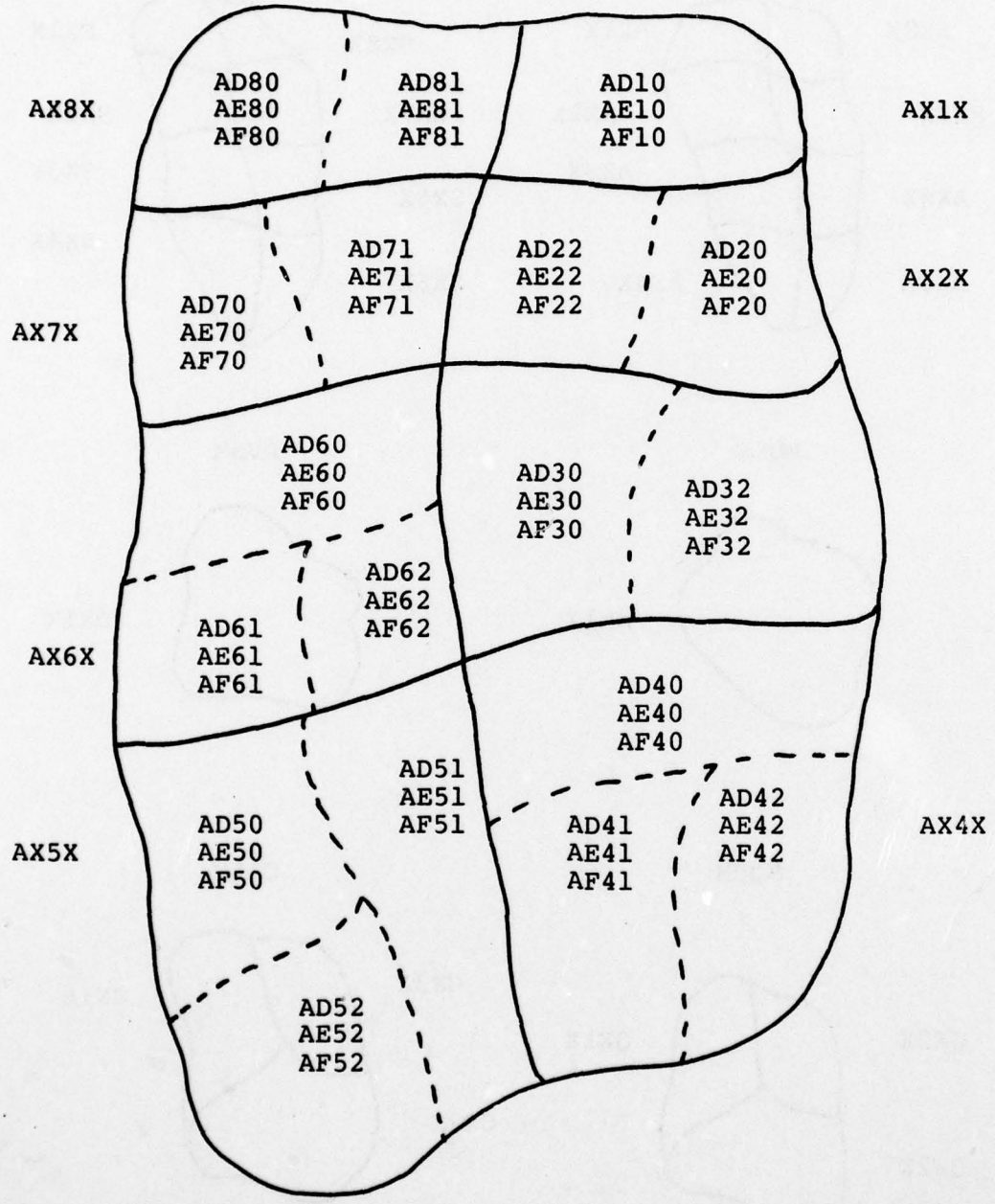


Figure C2 ADDITION SET WITH SUBSETS AND ELEMENTS IN GROUPS

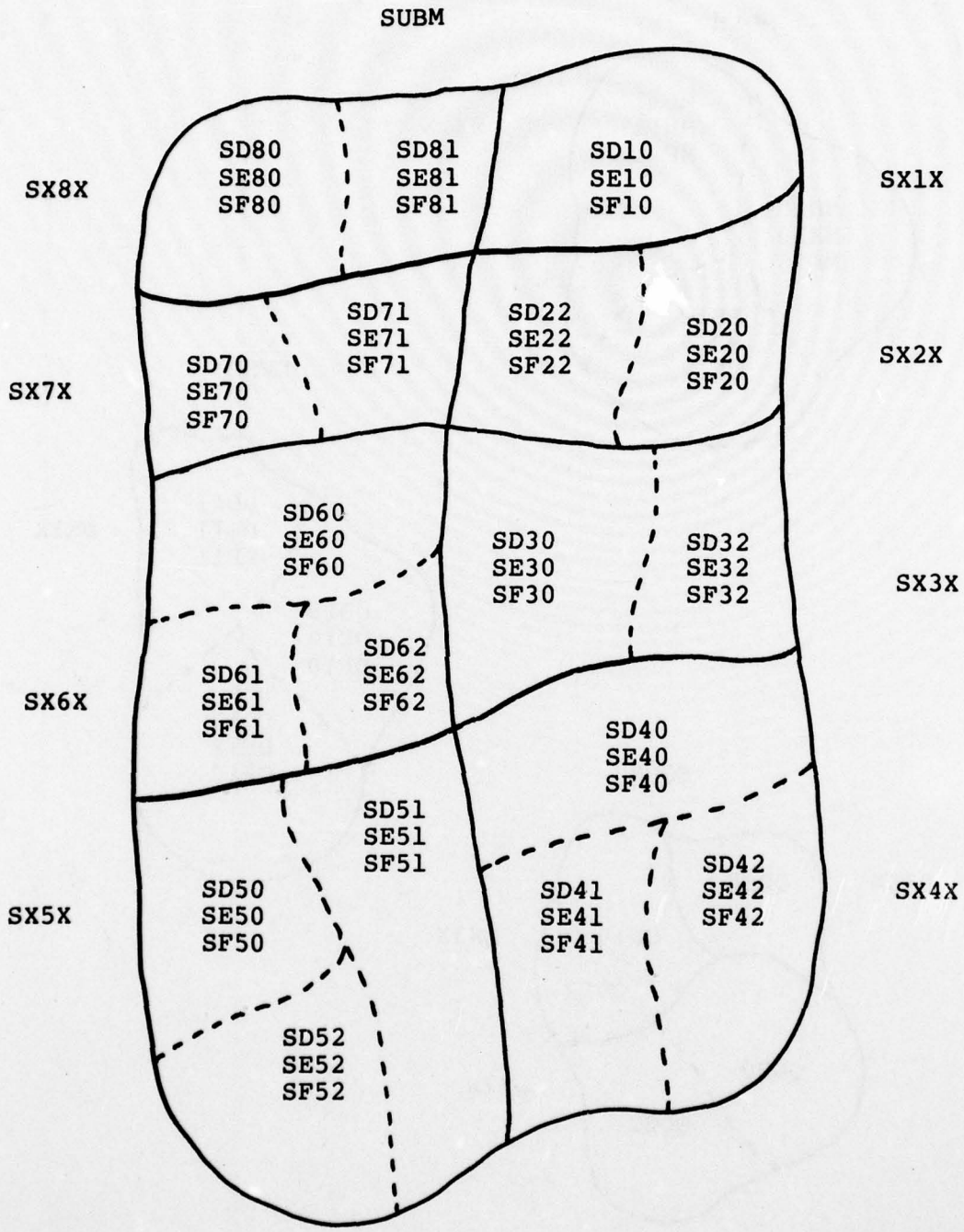


Figure C3 SUBTRACTION SET WITH SUBSETS AND ELEMENTS IN GROUPS

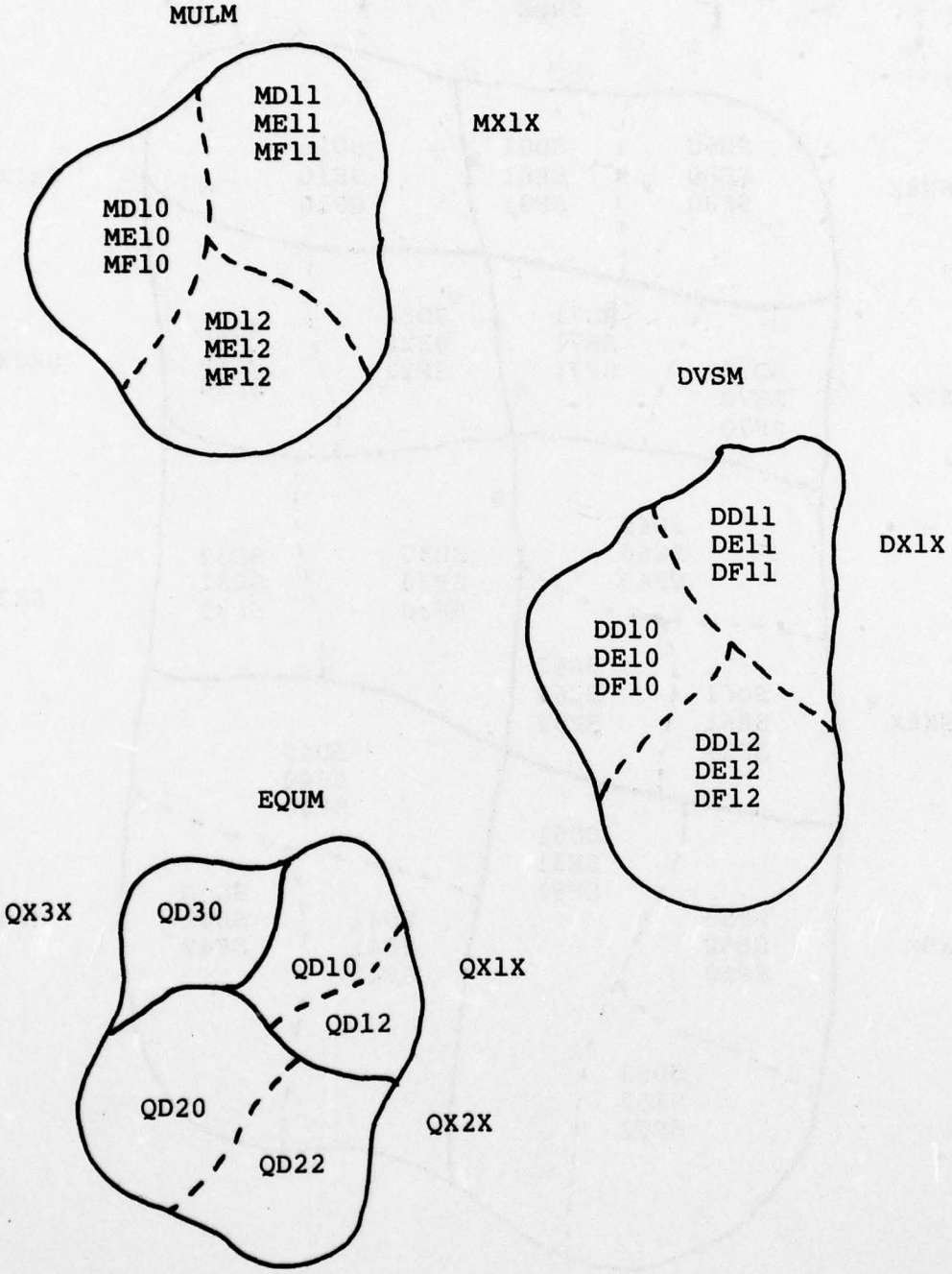


Figure C4 MULTIPLICATION, DIVISION, AND EQUALITY SETS WITH SUBSETS AND ELEMENTS IN GROUPS

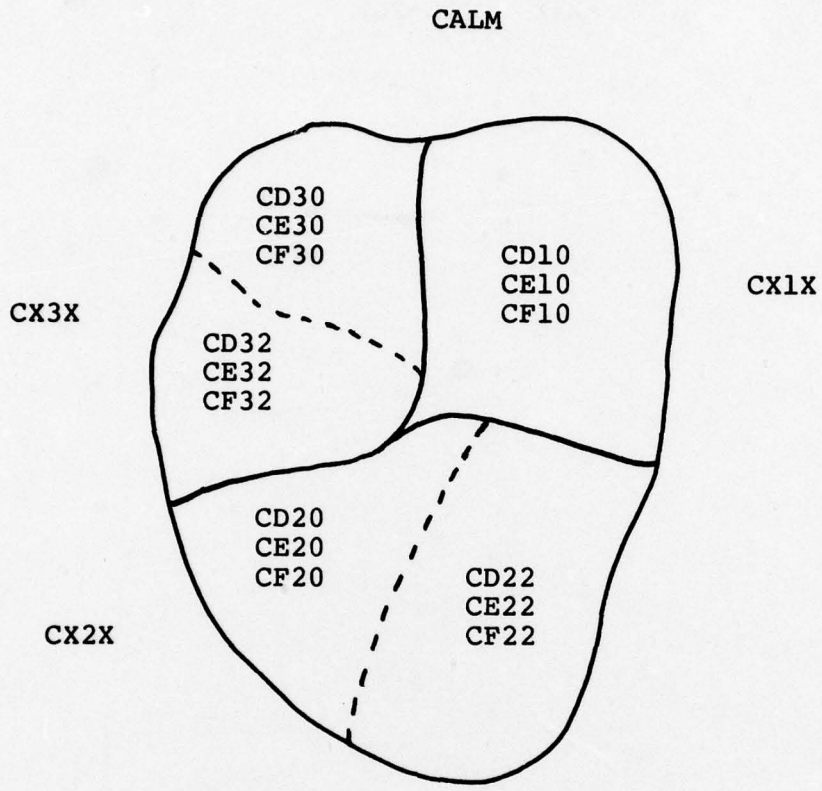


Figure C5 SUBROUTINE CALL SET WITH SUBSETS AND ELEMENTS IN GROUPS

ADDM

AX1X (1L-2L)

```
AD10
LDMQ    ARG1
QLS     PAR1
LDAC    ARG2
ALS     PAR2
AACQ
AQRS    PAR3
STMQ    ARG3


AE10
LDMQ    ARG1
QLS     PAR1
LDAC    ARG2
ALS     PAR2
AACQ
AQRS    PAR3
PMQA


AF10
LDMQ    ARG1
QLS     PAR1
LDAC    ARG2
ALS     PAR2
AACQ
AQRS    PAR3
```

AX2C (1L-2N)

AD20		AD22	
LDMQ	ARG1	LDAC	ARG1
QLS	PAR1	ALS	PAR1
ADMQ	ARG2	AACQ	
AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3

AE20		AE22	
LDMQ	ARG1	LDAC	ARG1
QLS	PAR1	ALS	PAR1
ADMQ	ARG2	AACQ	
AQRS	PAR3	AQRS	PAR3
PMQA		PMQA	

AF20		AF22	
LDMQ	ARG1	LDAC	ARG1
QLS	PAR1	ALS	PAR1
ADMQ	ARQ2	AACQ	
AQRS	PAR3	AQRS	PAR3

AX3X (1L-2R)

AD30		AD32	
LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2
LDAC	ARG1	LDAC	ARG1
ALS	PAR1	ALS	PAR1
AACQ		AACQ	
AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3

AE30		AE32	
LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2
LDAC	ARG1	LDAC	ARG1
ALS	PAR1	ALS	PAR1
AACQ		AACQ	
AQRS	PAR3	AQRS	PAR3
PMQA		PMQA	

AF30		AF32	
LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2
LDAC	ARG1	LDAC	ARG1
ALS	PAR1	ALS	PAR1
AACQ		AACQ	
AQRS	PAR3	AQRS	PAR3

AD-A059 041 NAVAL SURFACE WEAPONS CENTER WHITE OAK LAB SILVER SP--ETC F/G 9/2
A FIXED-POINT ARITHMETIC COMPILER.(U)
JUL 77 P CRAUN
UNCLASSIFIED NSWC/WOL/TR-77-65 NL

2 OF 2
AD
A059804

END
DATE
FILMED
11-78
DDC

AX4X (1N-2R)

AD40		AD41		AD42	
LDMQ	ARG2	LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2	AQRS	PAR2
ADMQ	ARG1	AACQ		ADMQ	ARG1
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3	STMQ	ARG3

AE40		AE41		AE42	
LDMQ	ARG2	LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2	AQRS	PAR2
ADMQ	ARG1	AACQ		ADMQ	ARG1
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
PMQA		PMQA		PMQA	

AF40		AF41		AF42	
LDMQ	ARG2	LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2	AQRS	PAR2
ADMQ	ARG1	AACQ		ADMQ	ARG1
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3

AX5X (1N-2N)

AD50		AD51		AD52	
LDMQ	ARG1				
ADMQ	ARG2	ADMQ	ARG2	ADMQ	ARG1
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3	STMQ	ARG3

AE50		AE51		AE52	
LDMQ	ARG1				
ADMQ	ARG2	ADMQ	ARG2	ADMQ	ARG1
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
PMQA		PMQA		PMQA	

AF50		AF51		AF52	
LDMQ	ARG1				
ADMQ	ARG2	ADMQ	ARG2	ADMQ	ARG1
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3

AX6X (1R-2N)

AD60		AD61		AD62	
LDMQ	ARG1			LDMQ	ARG1
AQRS	PAR1	AQRS	PAR1	AQRS	PAR1
ADMQ	ARG2	ADMQ	ARG2	AACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3	STMQ	ARG3

AE60		AE61		AE62	
LDMQ	ARG1			LDMQ	ARG1
AQRS	PAR1	AQRS	PAR1	AQRS	PAR1
ADMQ	ARG2	ADMQ	ARG2	AACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
PMQA		PMQA		PMQA	

AF60		AF61		AF62	
LDMQ	ARG1			LDMQ	ARG1
AQRS	PAR1	AQRS	PAR1	AQRS	PAR1
ADMQ	ARG2	ADMQ	ARG2	AACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3

AX7X (1R-2L)

AD70		AD71	
LDMQ	ARG1		
AQRS	PAR1	AQRS	PAR1
LDAC	ARG2	LDAC	ARG2
ALS	PAR2	ALS	PAR2
AACQ		AACQ	
AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3

AE70		AE71	
LDMQ	ARG1		
AQRS	PAR1	AQRS	PAR1
LDAC	ARG2	LDAC	ARG2
ALS	PAR2	ALS	PAR2
AACQ		AACQ	
AQRS	PAR3	AQRS	PAR3
PMQA		PMQA	

AF70		AF71	
LDMQ	ARG1		
AQRS	PAR1	AQRS	PAR1
LDAC	ARG2	LDAC	ARG2
ALS	PAR2	ALS	PAR2
AACQ		AACQ	
AQRS	PAR3	AQRS	PAR3
PMQA		PMQA	

AX8X (1N-2L)

```
AD80                        AD81
LDMQ    ARG2                LDAC    ARG2
QLS     PAR2                ALS     PAR2
ADMQ    ARG1                AACQ
AQRS    PAR3                AQRS    PAR3
STMQ    ARG3                STMQ    ARG3


AE80                        AE81
LDMQ    ARG2                LDAC    ARG2
QLS     PAR2                ALS     PAR2
ADMQ    ARG1                AACQ
AQRS    PAR3                AQRS    PAR3
PMQA                        PMQA


AF80                        AF81
LDMQ    ARG2                LDAC    ARG2
QLS     PAR2                ALS     PAR2
ADMQ    ARG1                AACQ
AQRS    PAR3                AQRS    PAR3
```

SUBM

SX1X (1L-2L)

```
SD10
LDMQ    ARG1
QLS     PAR1
LDAC    ARG2
ALS     PAR2
SACQ
AQRS    PAR3
STMQ    ARG3

SE10
LDMQ    ARG1
QLS     PAR1
LDAC    ARG2
ALS     PAR2
SACQ
AQRS    PAR3
PMQA

SF10
LDMQ    ARG1
QLS     PAR1
LDAC    ARG2
ALS     PAR2
SACQ
AQRS    PAR3
```

SX2X (1L-2N)

SD20		SD22	
LDMQ	ARG1	LDAC	ARG1
QLS	PAR1	ALS	PAR1
SBMQ	ARG2	SMQA	
AQRS	PAR3	PACQ	
STMQ	ARG3	AQRS	PAR3
		STMQ	ARG3

SE20		SE22	
LDMQ	ARG1	LDAC	ARG1
QLS	PAR1	ALS	PAR1
SBMQ	ARG2	SMQA	
AQRS	PAR3	PACQ	
PMQA		AQRS	PAR3
		PMQA	

SF20		SF22	
LDMQ	ARG1	LDAC	ARG1
QLS	PAR1	ALS	PAR1
SBMQ	ARG2	SMQA	
AQRS	PAR3	PACQ	
		AQRS	PAR3

SX3X (1L-2R)

SD30		SD32	
LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2
LDAC	ARG1	LDAC	ARG1
ALS	PAR1	ALS	PAR1
SMQA		SMQA	
PACQ		PACQ	
AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3

SE30		SE32	
LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2
LDAC	ARG1	LDAC	ARG1
ALS	PAR1	ALS	PAR1
SMQA		SMQA	
PACQ		PACQ	
AQRS	PAR3	AQRS	PAR3
PMQA		PMQA	

SF30		SF32	
LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2
LDAC	ARG1	LDAC	ARG1
ALS	PAR1	ALS	PAR1
SMQA		SMQA	
PACQ		PACQ	
AQRS	PAR3	AQRS	PAR3

SX4X (1N-2R)

SD40		SD41		SD42	
LDMQ	ARG2	LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2	AQRS	PAR2
SMBQ	ARG1	SMQA		SBMQ	ARG1
CHSQ		PACQ		CHSQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3	STMQ	ARG3

SE40		SE41		SE42	
LDMQ	ARG2	LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2	AQRS	PAR2
SBMQ	ARG1	SMQA		SBMQ	ARG1
CHSQ		PACQ		CHSQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
PMQA		PMQA		PMQA	

SF40		SF41		SF42	
LDMQ	ARG2	LDMQ	ARG2		
AQRS	PAR2	AQRS	PAR2	AQRS	PAR2
SBMQ	ARG1	SMQA		SBMQ	ARG1
CHSQ		PACQ		CHSQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3

SX5X (1N-2N)

SD50		SD51		SD52	
LDMQ	ARG1			LDMQ	ARG1
SBMQ	ARG2	SBMQ	ARG2	SACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3	STMQ	ARG3

SE50		SE51		SE52	
LDMQ	ARG1			LDMQ	ARG1
SBMQ	ARG2	SBMQ	ARG2	SACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
PMQA		PMQA		PMQA	

SF50		SF51		SF52	
LDMQ	ARG1			LDMQ	ARG1
SBMQ	ARG2	SBMQ	ARG2	SACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3

SX6X (1R-2N)

SD60		SD61		SD62	
LDMQ	ARG1			LDMQ	ARG1
AQRS	PAR1	AQRS	PAR1	AQRS	PAR1
SBMQ	ARG2	SBMQ	ARG2	SACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3	STMQ	ARG3

SE60		SE61		SE62	
LDMQ	ARG1			LDMQ	ARG1
AQRS	PAR1	AQRS	PAR1	AQRS	PAR1
SBMQ	ARG2	SMBQ	ARG2	SACQ	
AQRS	PAR3	AQRS	PAR3	AQRS	PAR3
PMQA		PMQA		PMQA	

SF60		SF61		SF62	
LDMQ	ARG1			LDMQ	ARG1
AQRS	PAR1	AQRS	PAR1	AQRS	PAR1
SBMQ	ARG2	SBMQ	ARG2	SACQ	
AQRS	PAR3	AQRS	PAR1	AQRS	PAR3
			PAR3		

SX7X (1R-2L)

SD70		SD71	
LDMQ	ARG1		
AQRS	PAR1	AQRS	PAR1
LDAC	ARG2	LDAC	ARG2
ALS	PAR2	ALS	PAR2
SACQ		SACQ	
AQRS	PAR3	AQRS	PAR3
STMQ	ARG3	STMQ	ARG3

SE70		SE71	
LDMQ	ARG1		
AQRS	PAR1	AQRS	PAR1
LDAC	ARG2	LDAC	ARG2
ALS	PAR2	ALS	PAR2
SACQ		SACQ	
AQRS	PAR3	AQRS	PAR3
PMQA		PMQA	

SF70		SF71	
LDMQ	ARG1		
AQRS	PAR1	AQRS	PAR1
LDAC	ARG2	LDAC	ARG2
ALS	PAR2	ALS	PAR2
SACQ		SACQ	
AQRS	PAR3	AQRS	PAR3

SX8X (1N-2L)

```
SD80                    SD81
LDMQ    ARG2            LDAC    ARG2
QLS     PAR2            ALS     PAR2
SBMQ    ARG1            SACQ
CHSQ                    AQRS    PAR3
AQRS    PAR3            STMQ    ARG3
STMQ    ARG3
```

```
SE80                    SE81
LDMQ    ARG2            LDAC    ARG2
QLS     PAR2            ALS     PAR2
SBMQ    ARG1            SACQ
CHSQ                    AQRS    PAR3
AQRS    PAR3            PMQA
PMQA
```

```
SF80                    SF81
LDMQ    ARG2            LDAC    ARG2
QLS     PAR2            ALS     PAR2
SBMQ    ARG1            SACQ
CHSQ                    AQRS    PAR3
AQRS    PAR3
```

MULM

MX1X (1L-2L)

MD10		MD11		MD12	
LDAC	ARG2			PMQA	
TACP	*+2	LDAC	ARG2	TACP	*+2
CHSA		TACP	*+2	CHSA	
ALS	PAR2	CHSA		ALS	PAR2
STAC	$V99	ALS	PAR2	STAC	$V99
LDAC	ARG2	STAC	$V99	PMQA	
LDMQ	ARG1	LDAC	ARG2	LDMQ	ARG1
EMQA		EMQA		EMQA	
ARS	23	ARS	23	ARS	23
SRAC	*+6	SRAC	*+6	SRAC	*+6
TMQP	*+2	TMQP	*+2	TMQP	*+2
CHSQ		CHSQ		CHSQ	
QLS	PAR1	QLS	PAR1	QLS	PAR1
MPY	$V99	MPY	$V99	MPY	$V99
PACQ		PACQ		PACQ	
PADA	**	PADA	**	PADA	**
TACZ	*+2	TACZ	*+2	TACZ	*+2
CHSQ		CHSQ		CHSQ	
STMQ	ARG3	STMQ	ARG3	STMQ	ARG3

MX1X (1L-2L)

```
ME10
LDAC    ARG2
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
LDAC    ARG2
LDMQ    ARG1
EMQA
ARS     23
SRAC    *+6
TMQP    *+2
CHSQ
QLS     PAR1
MPY     $V99
PACQ
PADA    **
TACZ    *+2
CHSQ
PMQA
```

```
ME11
LDAC    ARG2
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
LDAC    ARG2
EMQA
ARS     23
SRAC    *+6
TMQP    **2
CHSQ
QLS     PAR1
MPY     $V99
PACQ
PADA    **
TACZ    *+2
CHSQ
PMQA
```

```
ME12
PMQA
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
PMQA
LDMQ    ARG1
EMQA
ARS     23
SRAC    *+6
TMQP    **2
CHSQ
QLS     PAR1
MPY     $V99
PACQ
PADA    **
TACZ    *+2
CHSQ
PMQA
```

MX1X (1L-2L)

MF10		MF11		MF12	
LDAC	ARG2			PMQA	
TACP	*+2	LDAC	ARG2	TACP	*+2
CHSA		TACP	*+2	CHSA	
ALS	PAR2	CHSA		ALS	PAR2
STAC	$V99	ALS	PAR2	STAC	$V99
LDAC	ARG2	STAC	$V99	PMQA	
LDMQ	ARG1	LDAC	ARG2	LDMQ	ARG1
EMQA		EMQA		EMQA	
ARS	23	ARS	23	ARS	23
SRAC	*+6	SRAC	*+6	SRAC	*+6
TMQP	*+2	TMQP	*+2	TMQP	*+2
CHSQ		CHSQ		CHSQ	
QLS	PAR1	QLS	PAR1	QLS	PAR1
MPY	$V99	MPY	$V99	MPY	$V99
PACQ		PACQ		PACQ	
PADA	**	PADA	**	PADA	**
TACZ	*+2	TACZ	*+2	TACZ	*+2
CHSQ		CHSQ		CHSQ	

DVSM

DX1X (1L-2L)

```
DD10                    DD11                    DD12
LDAC    ARG2                                    PMQA
TACP    *+2             LDAC    ARG2            TACP    *+2
CHSA                    TACP    *+2             CHSA
ALS     PAR2            CHSA                    ALS     PAR2
STAC    $V99            ALS     PAR2            STAC    $V99
LDAC    ARG2            STAC    $V99            PMQA
LDMQ    ARG1            LDAC    ARG2            LDMQ    ARG1
EMQA                    EMQA                    EMQA
ARS     23              ARS     23              ARS     23
SRAC    *+6             SRAC    *+6             SRAC    *+6
TMQP    *+2             TMQP    **2             TMQP    *+2
CHSQ                    CHSQ                    CHSQ
ZAC                     ZAC                     ZAC
LLS     PAR1            LLS     PAR1            LLS     PAR1
DPDV    $V99            DPDV    $V99            DPDV    $V99
PADA    **              PADA    **              PADA    **
TACZ    *+2             TACZ    *+2             TACZ    *+2
CHSQ                    CHSQ                    CHSQ
STMQ    ARG3            STMQ    ARG3            STMQ    ARG3
```

DX1X (1L-2L)

```
DE10
LDAC    ARG2
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
LDAC    ARG2
LDMQ    ARG1
EMQA
ARS     23
SRAC    *+6
TMQP    *+2
CHSQ
ZAC
LLS     PAR1
DPDV    $V99
PADA    **
TACZ    *+2
CHSQ
PMQA
```

```
DE11
LDAC    ARG2
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
LDAC    ARG2
EMQA
ARS     23
SRAC    *+6
TMQP    *+2
CHSQ
ZAC
LLS     PAR1
DPDV    $V99
PADA    **
TACZ    *+2
CHSQ
PMQA
```

```
DE12
PMQA
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
PMQA
LDMQ    ARG1
EMQA
ARS     23
SRAC    *+6
TMQP    *+2
CHSQ
ZAC
LLS     PAR1
DPDV    $V99
PADA    **
TACZ    *+2
CHSQ
PMQA
```

DX1X (1L-2L)

```
DF10
LDAC    ARG2
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
LDAC    ARG2
LDMQ    ARG1
EMQA
ARS     23
SRAC    *+6
TMQP    *+2
CHSQ
ZAC
LLS     PAR1
DPDV    $V99
PADA    **
TACZ    *+2
CHSQ
```

```
DF11
LDAC    ARG2
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
LDAC    ARG2
EMQA
ARS     23
SRAC    *+6
TMQP    *+2
CHSQ
ZAC
LLS     PAR1
DPDV    $V99
PADA    **
TACZ    *+2
CHSQ
```

```
DF12
PMQA
TACP    *+2
CHSA
ALS     PAR2
STAC    $V99
PMQA
LDMQ    ARG1
EMQA
ARS     23
SRAC    *+6
TMQP    *+2
CHSQ
ZAC
LLS     PAR1
DPDV    $V99
PADA    **
TACZ    *+2
CHSQ
```

CALM

CX1X (2L-3R)

```
CD10
LDMQ    ARG2
QLS     PAR2
CALL    ARG1
AQRS    PAR3
STMQ    ARG3


CE10
LDMQ    ARG2
QLS     PAR2
CALL    ARG1
AQRS    PAR3
PMQA


CF10
LDMQ    ARG2
QLS     PAR2
CALL    ARG1
AQRS    PAR3
```

CX2X (2R-3R)

```
CD20                        CD22
LDMQ    ARG2
AQRS    PAR2                AQRS    PAR2
CALL    ARG1                CALL    ARG1
AQRS    PAR3                AQRS    PAR3
STMQ    ARG3                STMQ    ARG3


CE20                        CE22
LDMQ    ARG2
AQRS    PAR2                AQRS    PAR2
CALL    ARG1                CALL    ARG1
AQRS    PAR3                AQRS    PAR3
PMQA                        PMQA


CF20                        CF22
LDMQ    ARG2
AQRS    PAR2                AQRS    PAR2
CALL    ARG1                CALL    ARG1
AQRS    PAR3                AQRS    PAR3
```

CX3X (2N-3R)

```
CD30                    CD32
LDMQ    ARG2
CALL    ARG1            CALL    ARG1
AQRS    PAR3            AQRS    PAR3
STMQ    ARG3            STMQ    ARG3


CE30                    CE32
LDMQ    ARG2
CALL    ARG1            CALL    ARG1
AQRS    PAR3            AQRS    PAR3
PMQA                    PMQA


CF30                    CF32
LDMQ    ARG2
CALL    ARG1            CALL    ARG1
AQRS    PAR3            AQRS    PAR3
```

EQUM

QX1X (1L)

```
QD10
LDMQ    ARG1
QLS     PAR1
STMQ    ARG2
```

```
QD11
QLS     PAR1
STMQ    ARG2
```

QX2X (1R)

```
QD20
LDMQ    ARG1
AQRS    PAR1
STMQ    ARG2
```

```
QD21
AQRS    PAR1
STMQ    ARG2
```

QX3X (1N)

```
QD30
LDMQ    ARG1
STMQ    ARG2
```

SELECTED BIBLIOGRAPHY

1. Pryor, C.N., "Programming for the Digital Simulator and Computer (DISAC), "NOLTR 64-119, 1964.

2. Rosen, Saul, Programming Systems and Languages, McGraw-Hill Book Company, 1967.

3. Richards, R,K., Arithmetic Operations in Digital Computers, D. Van Nostrand Company, 1955.

4. Knuth, Donald, E., Fundamental Algorithms, Addison-Wesly Publishing Company, 1968.

5. Dimitry, Donald and Mott, Thomas, FORTRAN IV Programming, Holt, Rinehart, and Winston, 1966.

DISTRIBUTION

	Copies
Commander Naval Underwater Systems Center New London Laboratory New London, CT 02844 (Dr. C. N. Pryor)	1
Commanding Officer Naval Research Laboratory Washington, D. C. 20375 (Dr. Freeman, Code ONR-7590)	1
Intermetrics Incorporated 701 Concord Avenue Cambridge, Massachusetts 02138 (Dr. J. Pepe)	1
Defense Documentation Center Cameron Station Alexandria, Virginia 22314	12